Certified Compilation and Worst-Case Execution Time Estimation

André Oliveira Maroneze

ANNÉE 2014

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

André Oliveira Maroneze

préparée à l'Unité Mixte de Recherche 6074 - IRISA
Institut de recherche en informatique et systèmes aléatoires
UFR Informatique Electronique (ISTIC)

Verified Compilation and Worst-Case Execution Time Estimation

Thèse soutenue à Rennes
le 17 juin 2014

devant le jury composé de :

Björn LISPER
Professeur des universités - Mälardalen University / rapporteur

Andrew TOLMACH
Professeur des universités - Portland State University / rapporteur

Xavier LEROY
Directeur de recherche - INRIA Rocquencourt / examinateur

Marc PANTEL
Maître de conférences - ENSEEIHT / examinateur

Christine ROCHANGE
Professeur des universités - Univ. de Toulouse III / examinatrice

Sandrine BLAZY
Professeur des universités - Univ. de Rennes 1 / directrice de thèse

David PICHARDIE
Professeur des universités - ENS Rennes / encadrant de thèse

Isabelle PUAUT
Professeur des universités - Univ. de Rennes 1 / co-directrice de thèse

Contents

Résumé en français

Contexte

Nous dépendons de plus en plus des systèmes informatiques dans nos vies : pour le loisir, la communication, le travail, mais aussi à l'intérieur de systèmes dits *critiques* : systèmes médicaux, nucléaires, transports (avions, trains, voitures), etc. Dans ces systèmes, une erreur logicielle peut entraîner des coûts très élevés, en termes matériels ainsi qu'en vies humaines. Un niveau très élevé de sureté est exigé dans ces systèmes, ainsi bien du côté matériel qu'en termes de logiciel. Le processus de développement de ces logiciels passe par plusieurs étapes afin d'obtenir les niveaux les plus élevés de garantie en termes de robustesse et absence d'erreurs.

Dans le domaine très large du génie logiciel, qui étudie les processus et méthodes de développement logiciel pour augmenter sa qualité et maîtriser son coût, nous retrouvons les *méthodes formelles*, un ensemble de connaissances et techniques utilisées pour partir des spécifications fonctionnelles (correspondant au comportement souhaité des systèmes) et aboutir à des logiciels dont le comportement est conforme à celui attendu. Parmi les plusieurs familles de méthodes formelles, nous nous intéressons plus particulièrement à la *vérification formelle*, qui consiste à prouver, à l'aide d'une description précise du comportement d'un programme, qu'il respecte sa spécification (celle-ci est aussi formulée de manière précise). Ce niveau de garantie est parmi les plus élevés que l'on puisse obtenir en termes d'assurance du logiciel. Il a cependant un coût élevé, raison pour laquelle il n'est utilisé que dans des systèmes critiques.

Systèmes temps-réel et temps d'exécution au pire cas

Une caractéristique commune à la plupart des systèmes critiques est son caractère *temporel* : parce qu'ils interagissent avec le monde externe, ces systèmes ont de fortes contraintes en termes de temps de calcul et réaction, qui sont souvent peu présentes ou moins strictes dans d'autres systèmes informatiques. Par exemple, si un logiciel de bureautique prend 1 ou 10 secondes à réagir à une entrée de son utilisateur, le résultat obtenu sera toujours le même (modulo quelques légers inconvénients) ; cependant, si un aileron dans un avion, ou un scintigraphe médical subissent du retard dans leur activation, le système physique avec lequel ils interagissent se comportera de façon très différente. Dans ce type de système, appelé *temps-réel*, la correction du logiciel dépend non seulement du résultat calculé mais aussi du temps pris pour l'obtenir. L'application de méthodes formelles dans le développement de logiciels pour de systèmes temps-réel critiques est une nécessité incontournable, et des techniques pour vérifier ce type de logiciels constituent un enjeu scientifique majeur.

Un aspect très étudié dans la vérification des systèmes temps-réel est le *temps d'exécution au pire cas*, communément appelé WCET de par l'abréviation du terme en anglais (*Worst-Case Execution Time*). Dans plusieurs systèmes, ce temps constitue une mesure suffisante pour garantir le bon fonctionnement temporel du système. Cette propriété ne peut cependant pas être obtenue, dans le cas général, de manière exacte. Nous cherchons donc une sur-approximation de cette valeur (ce qui garantit la sûreté du système), aussi proche que possible de sa valeur exacte. Pour cela, nous pouvons appliquer des techniques sophistiquées d'analyse de programmes. Le résultat, combiné à un choix d'ordonnancement des différentes activités du système, indique s'il peut ou non répondre aux besoins temporels exigés.

Compilation

Un aspect important du développement logiciel consiste dans la traduction du code écrit par le programmeur, dans un langage de programmation adapté à l'écriture et à l'analyse par des humains, vers du code exécutable par la machine, plus éloigné des spécifications et moins facilement analysable par des humains. Ce processus de traduction, appelé *compilation*, qui incorpore aussi des étapes d'amélioration du code (dites *optimisations*), bien que fondé sur des bases théoriques assez élaborées, est sujet à des erreurs dû à sa complexité. Ces erreurs, introduites à l'intérieur du processus de compilation et d'optimisation, rendent peu fiable la vérification des spécifications au niveau du code source, car le code cible, qui est celui vraiment exécuté par le système, peut ne pas lui correspondre. Il en résulte que la vérification du système, pour être fiable, doit se faire dans le code machine, ce qui augmente la complexité et le coût de la vérification.

L'utilisation d'un compilateur formellement vérifié, exempt d'erreurs de traduction, permet de remonter l'analyse au niveau du langage source. En outre, le processus de compilation lui-même permet d'obtenir des propriétés liées au code qui peuvent être utiles pour d'autres analyses, telles que celle liée à l'estimation du temps d'exécution au pire cas (WCET). La combinaison entre compilation et estimation du WCET est une voie de recherche active, et le sujet central de cette thèse, plus particulièrement la vérification formelle de techniques utiles pour l'estimation du WCET, intégrées dans un compilateur formellement vérifié.

Techniques dérivées

Plusieurs méthodes d'estimation du WCET sont basées sur une combinaison de différentes techniques d'utilisation plus générale, telles que le *découpage de programmes* (*program slicing*), l'interprétation abstraite et la résolution de systèmes de programmation linéaire. Chacune de ces techniques peut être adaptée à d'autres contextes et utilisée de manière plus ou moins indépendante. Ces techniques dérivées constituent aussi une partie importante de la formalisation de l'estimation du WCET et peuvent servir dans d'autres contextes. Elles ne sont pas l'objectif premier de cette thèse, mais constituent des contributions secondaires significatives.

Objectifs

Les objectifs principaux de cette thèse sont :

- l'*obtention de garanties formelles* sur une méthode d'estimation du WCET basée sur l'état de l'art ;
- l'*intégration de ces techniques dans un compilateur réaliste*, pour le langage C (utilisé dans l'industrie, à l'inverse de langages "jouets"), et l'évaluation expérimentale des résultats.

L'objectif à long terme est d'utiliser les résultats de ces travaux pour obtenir une estimation de WCET entièrement vérifiée, intégrée à un compilateur lui aussi vérifié (en occurrence, le compilateur C CompCert) adapté à des systèmes temps-réel critiques. Cela nécessite, entre autres, une formalisation de l'architecture matérielle (y compris des composantes telles que la mémoire cache et le *pipeline*), ainsi que des étapes d'estimation du WCET liées à ces éléments.

Contributions

Les principales contributions de cette thèse sont présentées par la suite.

Estimation de bornes de boucles pour le WCET

Une première contribution est la formalisation et évaluation expérimentale d'une méthode d'estimation de bornes de boucles, applicable entre autres au contexte de l'estimation du WCET. Les garanties formelles apportées permettent d'établir des bornes supérieures sur le nombre d'exécutions de chaque point du programme en langage assembleur, obtenu après compilation. Ce programme est

celui qui sera utilisé par les outils d'estimation de WCET, et les bornes fournies permettent d'éviter l'utilisation d'annotations manuelles.

Cette estimation de bornes de boucles, formalisée dans le chapitre 4, dépend de deux techniques qui sont liées à d'autres contributions de cette thèse (découpage de programmes et analyse de valeurs). L'évaluation expérimentale permet de vérifier que note méthode formalisée obtient des résultats proches à ceux de la méthode SWEET [35] de l'état de l'art, qui a servi d'inspiration à notre méthode.

Découpage de programmes

La deuxième contribution de cette thèse est la formalisation, le développement et la vérification du *découpage de programmes* (*program slicing*), partie intégrante d'une analyse de bornes de boucles. Le chapitre 5 est dédié à cette contribution.

Ce découpage de programmes, effectué sur un des langages intermédiaires du compilateur CompCert, est applicable à d'autres méthodes que l'estimation de bornes de boucles. Le calcul du découpage est effectué par du code non vérifié et ensuite le résultat est validé *a posteriori*. Ce découplage permet l'utilisation d'algorithmes efficaces pour le calcul du découpage.

Évaluation d'une analyse de valeurs

La troisième contribution est l'évaluation expérimentale d'une analyse de valeurs certifiée [11]. Une analyse de valeurs calcule l'ensemble des valeurs possibles pour chaque variable, à chaque point de programme. Le chapitre 6 détaille une analyse de valeurs effectuée dans le cadre du développement d'un analyseur statique pour des programmes C. La partie expérimentale de cette analyse inclut une comparaison avec d'autres outils de la littérature d'analyse de valeurs pour du code C. L'évaluation permet de vérifier que les résultats calculés par l'analyse sont non-triviaux.

Formalisation papier de la méthode de calcul de WCET basée sur IPET (*Implicit Path Enumeration Technique*)

Une quatrième contribution de cette thèse est la formalisation, cette fois-ci sur papier (en ayant comme objectif la vérification mécanique), d'une technique répandue d'estimation de WCET, l'IPET [49]. Cette formalisation est le sujet de la section 7.1. Cette technique se base sur la génération d'un programme linéaire entier représentant les possibles flots de contrôle du programme, et l'obtention du WCET comme solution maximale du système.

Notre formalisation de la méthode IPET inclut la génération de contraintes linéaires à partir du programme assembleur et à partir des bornes de boucles calculées préalablement, et aussi la vérification *a posteriori* du résultat fourni par un solveur externe. Cette vérification, à partir d'un *certificat de Farkas*, permet de s'affranchir de l'utilisation d'un solveur linéaire vérifié. Le résultat certifié est l'estimation formellement vérifiée du WCET.

Implémentation et évaluation d'un ensemble d'outils d'analyse statique

La dernière contribution de cette thèse est l'implémentation d'un ensemble d'outils d'analyse statique dérivés des développements formels et évaluations expérimentales effectués dans cette thèse. Cet ensemble d'outils peut être utilisé dans le développement et évaluation de plusieurs analyses statiques.

Décrits dans la section 7.2, ces outils offrent des fonctionnalités complémentaires au développement formel, telles que des transformations de programme liées à la compilation (par exemple, inversion et déroulage de boucles), un interprète de référence pour le langage assembleur, en plus de toutes les analyses présentées précédemment.

L'évaluation expérimentale de deux de ces outils, l'analyse de bornes de boucles et l'estimation de WCET, est présentée dans la section 7.3. Cette évaluation compare la précision de nos outils vérifiées à celle d'un outil de référence (non formellement vérifié) dans le domaine du WCET. Via un ensemble de *benchmarks* de référence, nous effectuons une comparaison de la précision des outils. Les résultats indiquent que la précision des nos outils est satisfaisante.

Conclusion

Nous avons formalisé les étapes majeures d'un outil d'estimation de WCET, depuis l'analyse de flot (avec l'estimation de bornes de boucles) jusqu'à l'obtention d'un WCET via la méthode IPET. Notre méthode considère un modèle matériel simple qui peut être étendu à des architectures plus réalistes. Notre outil est intégré au compilateur formellement vérifié CompCert et il peut traiter des programmes C, qui sont compilés vers du code assembleur PowerPC.

Notre formalisation comprend plusieurs étapes placées à des niveaux différents de la chaîne de compilation, ce qui montre l'intérêt de développer une analyse de WCET intégrée à un compilateur. Les garanties formelles apportées par notre méthode contribuent à l'évolution de la chaîne de traitement certifié pour le développement de systèmes critiques.

Le développement associé aux multiples évaluations expérimentales a confirmé que la formalisation d'un outil complexe exige des précautions pour la minimisation de l'effort de preuve, pour qu'elle reste faisable dans un temps raisonnable. Aussi, nous avons confirmé qu'au lieu d'effectuer la formalisation de façon indépendante de l'évaluation et du test, un développement en parallèle permet de simplifier les preuves et les outils. Enfin, l'expérience a montré que l'utilisation de validation *a posteriori* est essentielle à des endroits où la preuve formelle est trop coûteuse, comme par exemple dans l'utilisation d'un solveur linéaire non vérifié.

Les perspectives à court terme sont la vérification mécanique de l'estimation de WCET via l'IPET et l'incorporation d'un modèle matériel plus sophistiqué. À moyen terme, l'introduction d'autres analyses pour l'augmentation de la précision de l'estimation (par exemple, des analyses de cache et pipeline, ou des analyses liées à la mémoire) est envisageable. Cela permettrait également de traiter plus de programmes, grâce à un plus grand nombre de boucles bornées automatiquement. À long terme, nous envisageons la vérification des outils plus en aval sur la chaîne de développement de logiciel critique, pour travailler à des niveaux plus proches du matériel et obtenir une meilleure précision. L'apport de plus de garanties formelles, aussi bien pour le compilateur que pour les outils associés, est un moyen sûr pour augmenter la confiance dans les systèmes critiques.

Chapter 1

Introduction

Computer systems are present in our everyday lives, both as tools for productivity and leisure, and as components of complex, critical systems, such as transportation and power generation. Nevertheless, not all computer systems are created equal: a programming error in a television set-top box usually has mild consequences, such as preventing us from watching our favorite TV series, but an error in a flight control system can quickly lead to the loss of human lives and costly equipment. In *safety-critical systems*, errors are expensive and subject of much attention: nuclear power generation, medical imaging equipment and transportation are typical examples of such systems.

To minimize the possibility of errors, hardware and software engineering techniques used in safety-critical systems systems are increasingly based on *formal methods*, with the application of mathematically sound principles and techniques to verify that the actual behavior of the system is conform to its specification, that is, its expected behavior. In this thesis, we are particularly interested in a kind of formal method called *formal verification*, based on mechanically verified proofs. In traditional, pen-and-paper proofs, the reader needs to manually verify or trust each line of the proof. Mechanical proofs, however, can be read and checked by an algorithm. This algorithm still needs to be manually checked, but only once: afterwards, all proofs can be verified by the same algorithm. Also importantly, writing a mechanical proof helps the proof writer to identify and eliminate ambiguities and implicit reasoning, since otherwise the computer will not accept the proof. Such mechanical verification enables a very high level of confidence in the system.

The proofs we consider here are correctness proofs of program transformations within a compiler. These transformations include the passage from high-level specifications to low-level code, and also analysis-related transformations (e.g. program simplifications which enable analyses to be more precise). These proofs are based on *formal semantics* of the programming languages involved in the transformation. A formal semantics for a given language is a mathematical description of the behavior of any program written in that language, independently of any implementation details.

Formally specifying the semantics of a programming language, especially for an industrial-strength language, such as C, is a lengthy process. It is only recently that formal verification has become sufficiently mature to be applied to industrial-scale tools based on these languages. While there are increasingly more applications of formal verification, such as a compiler (the CompCert C compiler [16]), an operating system microkernel (seL4 [38]) and a CPU simulator [65], formal verification remains an active area for research and development. However, important tools for the development of safety-critical software are still needed, and the work in this thesis contributes to such formalizations.

Most safety-critical systems are *real-time* systems: their correctness depends not only on computing the expected results, but on producing them *within given time constraints*. In particular, embedded[1] systems are often subject to physical constraints (e.g. due to sensors and actuators) that imply timing dependencies which make them real-time systems. While occasional lag and jitter are commonplace in desktop applications and Internet browsers, in safety-critical systems such as fly-by-wire software, even a small delay can cause loss of vehicle control. Such issues, which

[1] The term *embedded* emphasizes that the computer is part of another system, often a *cyber-physical* one, with interactions between computational and physical processes.

are due to both hardware and software elements, are studied in real-time systems analysis. In this thesis, we consider a specific property of real-time systems, namely the *worst-case execution time* (WCET), which is the maximum time taken by a program before it finishes execution, considering all of its possible executions (i.e. with different inputs). For instance, if the program in question is the computation of the response to a fly-by-wire command, then its WCET is the upper bound on how long it will take before the response is actually issued. The WCET is necessary for task scheduling, and for determining whether a given hardware and software configuration satisfies the timing constraints of the system in question.

It is not possible to compute the exact WCET for every program: in industrial-size applications, there are too many possibilities to consider. Running a program to measure its execution time only provides information about the specific input values used in that execution. Testing all possible input values is impracticable. *Static analysis* techniques provide a more efficient approach to this problem: they allow us to obtain results valid for *every* program execution, without having to run them. However, they are not exact: they compute *over-approximations* of the actual WCET. This requires extra hardware resources, but it is nevertheless acceptable, since the actual WCET cannot exceed the result of the static analysis, and therefore deadlines will still be respected. Static analysis examines the syntax of a program and infers properties valid for every execution. Sophisticated analyses produce more precise results, which in our case (WCET estimation) minimize underutilization of the hardware. However, as any component in the development chain of a safety-critical system, static analyses are themselves subject to incorrect specifications and implementation errors. It is necessary to formally verify these analyses to obtain confidence in their result. To do so, we must look in more detail at the architecture of such analyses.

Static analyses for WCET estimation are usually performed at the binary level. However, some high-level information available at the source code is lost during compilation (e.g. variable signedness and typing), and this information can improve the precision of the analysis. Also, some analyses are naturally defined in terms of more abstract code representations (for instance, with structured loops and expressions), and adapting them to work at the binary level requires contrived workarounds. To account for these facts, a solution is the integration of a compiler in the WCET estimation process. In a formally verified development, however, this requires a compiler that is itself verified, such as the CompCert [16] C compiler. CompCert is equipped with a semantic preservation theorem which ensures that its compiled code behaves like its corresponding source code. In other words, we can perform analyses at the source level and compute properties which are guaranteed to remain valid at the binary level.

The work in this thesis concerns the development of a formally verified static WCET estimation tool. We specify, implement and prove correct this tool, based on the formal framework provided by the CompCert compiler. The objective is to improve confidence in safety-critical systems development, by providing tools operating over a language as expressive as C, with associated correctness proofs and experimental evaluations of their results.

This thesis stands in the intersection between three areas: real-time systems, formal methods and compilation (as illustrated in Figure 1.1). We claim that *it is feasible to obtain semantic guarantees about WCET estimations based on state-of-the-art methods, and to integrate this estimation in a formally verified compiler which shares the same semantics.* The end result is a compiler with stronger guarantees related to WCET estimation, useful for safety-critical systems.

1.1 A Formally Verified, Static WCET Estimation Tool

The approach "implement first, then prove later" is not typically used in verified software development, especially for substantial formalizations. In particular, it is hard to take an existing implementation, developed without concern about mechanical verification, and prove it correct: the effort necessary to do so would be greater than re-implementing the algorithm in a proof-friendly way. The preferred approach in this domain, which we follow here, is the *software-proof co-design*: specification and proof are developed together. It is a realistic approach because, when proving the correctness of a software, developers often realize that some changes to the specification widely contribute to the proofs. For this reason, the architecture of a verified development is considerably different from that of a non-verified development.

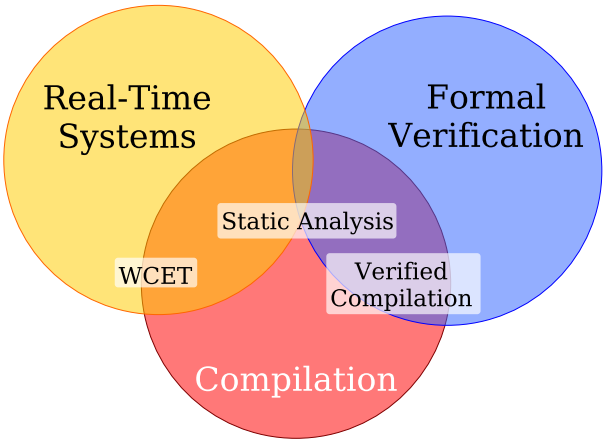


Figure 1.1: The three main computer science fields related to this thesis, along with topics related to this thesis.

To prove the correctness of our WCET estimation tool, we use the Coq [21] interactive *proof assistant*. A proof assistant allows the specification of algorithms, the definition of properties about such algorithms (such as correctness with respect to a given semantics), and the writing of proofs showing the validity of these properties. Proofs are written step-by-step, interactively with the help of the assistant. At each step, it displays the current goal and verifies if the next proof step given by the user is valid, providing fast feedback. To do so, proof assistants such as Coq include a proof-checking mechanism responsible for the mechanical verification. In some cases, the proof assistant can also help to obtain executable code, if the specification is executable (i.e. non-relational) and all program transformations terminate. This mechanical code generation is guaranteed to respect the specification and it improves confidence in the implementation.

The work in this thesis is split into several separate components of the WCET estimation tool. One such component is an automatic loop bound estimation method for C programs, which is itself composed of several independent techniques: program slicing, value analysis and bounds computation. The formal verification of these techniques constitutes the main contribution of this thesis.

- Program slicing consists in simplifying a program (by removing some unused statements) with respect to some criteria. In our case, the criteria are loop conditions in the program: by slicing the program one loop at a time, we improve the precision of our loop bound estimation method. In theory, program slicing is an optional step, since it is just an optimization for our estimation. In practice, however, it is essential for precision: without program slicing, the estimated bounds would be too loose. Proving that a slice is correct is quite a challenge, because it is an aggressive transformation with complex invariants.

- A value analysis computes all possible values for each program variable, at each program point. Like program slicing, it has applications in several domains. In our WCET estimation tool, the result of the analysis serves as input to the loop bound computation: loops with few iterations have variables whose values do not vary much. Both precision and proof effort of a value analysis are proportional to its sophistication. Realistic programs require fairly sophisticated analyses to obtain precise results.

- The last stage of the loop bound estimation is a computation step which produces bounds for each loop and combines them in the presence of nested loops, whose bounds are multiplied by the bounds of their enclosing loops. The challenge here is to prove that this composition results in a semantically correct estimation. This integration exemplifies a situation encountered in formal developments, where an intuitively simple idea requires a sophisticated formal reasoning. In the end, we obtain formally verified loop bounds for our program, but not yet the actual WCET estimation.

The WCET estimation itself is performed using the *Implicit Path Enumeration Technique* (IPET), commonly applied in state-of-the-art WCET estimation tools. This technique models control flow in the program through linear relations between the number of executions of different program points and/or program transitions (i.e. control flow graph vertices and edges), such as: "*the sum of executions of each* `if` *branch is equal to the number of executions of the* `if` *condition*". This produces a system of linear inequalities whose solutions correspond to different execution flows in the program. If we add coefficients corresponding to the number of clock cycles taken by each instruction (that is, a *hardware timing model*, mapping assembly instructions to their execution time), then the maximal solution of this system (that is, the sum of the executions of each program point multiplied by its timing coefficient) corresponds to the worst-case execution time estimation. The shape of these linear inequalities depends on the loop bounds previously obtained and on the timing model.

Proving the correctness of IPET via standard proof techniques would be prohibitively costly: it would include the formalization of an integer linear solver, which is beyond the scope of this thesis. Instead, we adopt *a posteriori validation*, a proof technique which consists in checking, or validating, that the result of a given algorithm (in our case, IPET) is correct, without having to formalize and prove each intermediate step of the computation. Because validation is performed on a single program, it is sometimes simpler than verifying that the algorithm is correct for *every* program. Notice that, while having a validator by itself does not ensure correctness, the validator can be formally verified itself. In a verified validator, every result that passes validation is provably correct. This technique is also used in other parts of our formal development, and it provides a cost-effective way to verify properties whose computation relies on complex algorithms. In particular, we argue that the formal verification of sophisticated software requires the use of both verified validation and traditional proof as complementary techniques.

The final result is a formally verified, executable tool integrated into the CompCert compiler. Extensive experimental evaluation of the tool has been performed, to assess the precision and efficiency of the analyses. Concerning the loop bound estimation, it has been compared to the non-verified implementation which served as inspiration for our formalization, using a set of reference WCET benchmarks. Concerning the value analysis, a more complex evaluation mechanism has been set in place, due to the fact that there are no reference benchmarks in this domain; it has been necessary to choose representative benchmarks, find comparable value analysis tools, define meaningful criteria for the comparison, and evaluate them. Finally, concerning the WCET estimation, its precision has been measured by comparing it to the ideal result, obtained via an interpreter of the assembly language. This interpreter is based on the formal semantics of the language, which ensures its result is exact.

In the following, we present in more detail the contributions of this thesis, as well as an outline of the content in each chapter.

1.2 Contributions

The contributions in this thesis include formal specifications, implementations and experimental evaluations. We highlight here the parts developed in this thesis which contribute to the state-of-the-art.

1. Formal development and experimental evaluation of a loop bound estimation for C programs. This estimation is based on a state-of-the-art tool for WCET estimation, SWEET [35]. The loop bound analysis is the first stage of our WCET estimation. It has been evaluated on the Mälardalen WCET benchmarks [34] and compared to SWEET in [12]. This work is presented in Chapter 4.

2. An executable formalization of program slicing, including specification and proof of correctness. It is used as part of the loop bound estimation (contribution 1). This contribution is the subject of Chapter 5, and this work has also been published in [12].

3. An experimental evaluation and benchmarking of a C value analysis. The analysis has been developed as a team effort for the Verasco ANR research project, and I focused on

the evaluation and benchmarking of the analysis, defining criteria and a methodology to compare it with other value analyses found in the state-of-the-art. This work constitutes the experimental evaluation of [11], and the value analysis is presented in Chapter 6.

4. A pen-and-paper formalization of an Implicit Path Enumeration Technique (IPET), described in Section 7.1, which has been applied in our WCET estimation, and the definition of a uniform framework integrating the aforementioned WCET-related analyses (presented as a tool in Section 7.2) for the analysis of programs compiled from C sources. Besides loop bound and WCET estimation, it performs (as independent analyses) program slicing and value analysis.

A detailed table containing the persons involved in each part of the development is presented in Appendix C. The entire formal development in Coq for all contributions is available online at:
http://www.irisa.fr/celtique/maroneze/phd.html
It can be downloaded and run with the corresponding versions of Coq (8.3) and OCaml (4.00 or newer). The development includes the modified source code of CompCert 1.11, which has been used in this thesis. Most theorem statements in this document contain the text [Coq Proof] at the end, which in the electronic version is a hyperlink to the proof in the website.

1.3 Thesis Outline

We describe here the content of each chapter in this thesis, which is composed of two main parts: the first one includes this introduction, the context and general definitions used throughout the manuscript, and a review of the state-of-the-art. It comprises chapters 1 to 3. The second part of this thesis, from chapters 4 to 7, is dedicated to each contribution, including technical details about the development and the proofs. Chapter 8 concludes the thesis.

An overview of formal verification and compilation (including important concepts and tools used in this thesis) is presented in Chapter 2. The beginning of the chapter focuses on the tools and techniques used for the proof (i.e. *how* the formalization was done, instead of *what* was formalized), such as the Coq proof assistant and the technique of *a posteriori* validation. Then, we describe succinctly the CompCert C compiler and some of its aspects relevant to our development, such as its annotation mechanism and the intermediate language used in our analyses.

Regarding the context about WCET estimation and static analysis, Chapter 3 details one of the most commonly used architectures for static WCET estimation, which is the basis of our method. It incorporates a value analysis based on *abstract interpretation*, a framework suitable for defining and proving static analyses. We present the main concepts related to this framework, thus providing a primer on abstract interpretation. Finally, we present a survey on state-of-the-art techniques and methods related to WCET estimation, static analysis and compilation.

The detailed formalizations and contributions begin in Chapter 4, with the description of our loop bound estimation. We follow a top-down approach, presenting the overall architecture before detailing each part. Some concepts presented here are used throughout the thesis. For instance, we introduce *execution counters* in our formal semantics, used to prove that the subcomponents of the loop bound estimation are correct. We then define loop bounds, and afterwards we present a precise description of the loop bound estimation algorithm and its correctness proof.

Chapter 5 deals with all aspects related to program slicing in this thesis. It consists of three main parts: the slicing algorithm, its proof of correctness, and the application of program slicing, that is, its integration in the loop bound estimation.

Value analysis is the subject of Chapter 6. Despite being used as a component of our loop bound estimation, in this chapter we describe a more general value analysis for C programs, whose long-term goal is to prove the absence of undefined behaviors in C programs for safety-critical systems. It is an evolution of the value analysis developed for our loop bound estimation. The chapter is divided in two parts. First, it describes the value analysis itself, including its several abstract domains. Then, we present the empirical evaluation we have conducted: a comparison, using CompCert's set of benchmarks, of the results obtained by (a) our value analysis, (b) Frama-C's [19] interval analysis, and (c) Navas' [54] range analysis. All of these analyses have been developed for C code, and their experimental evaluation is a contribution of this thesis.

Going beyond loop bound estimation, we use the IPET approach and a simplified hardware cost model to generate an ILP system and obtain our WCET estimation. In Chapter 7, we present the pen-and-paper formalization of this method. The IPET operates at the assembly level and reuses the result of the loop bound estimation to produce a provably correct estimation. Afterwards, we detail the implementations developed in this thesis, united into a single tool as an extension of CompCert. We conclude the chapter with two experimental evaluations, one for the loop bound estimation (which implicitly evaluates the program slicing and the value analysis), and another for the WCET estimation itself, where we performed some experiments to improve its precision. We evaluate our loop bound estimation on a set of reference WCET benchmarks, commonly referred to as the *Mälardalen WCET benchmarks*. The precision of the WCET estimation, for programs where the worst-case path is known, is computed by comparing it to a *reference interpreter* of CompCert's assembly language. This reference interpreter, based on the formal semantics of the PowerPC assembly, is also a contribution of this thesis.

Finally, Chapter 8 concludes the thesis, summarizing the results for each of the subjects involved in this work. Ideas for extensions, improvements and further work are presented at the end of that chapter.

Chapter 2

Context: Formal Verification and Compilation

This thesis is concerned with the links between WCET estimation, compilation and formal verification. In this chapter, we present the context related to the formal verification aspects of this thesis. In Section 2.1, we introduce the main concepts related to formal program verification, more specifically to machine-checked proofs of program correctness. Then, in Section 2.2, we introduce the CompCert C compiler, which is a formally verified compiler that will serve as framework for the integration of our WCET-related analyses. Correctness of our techniques is established with respect to to CompCert's semantics, and all throughout the thesis we use concepts related to one of its intermediate languages, Cfg (Section 2.2.4), which are detailed in this chapter.

2.1 Machine-Checked Proofs

A machine-checked proof comprises two parts: on one side, the *specification* of some theorem, lemma, or behavior, to which we associate a meaning. On the other, a *proof script*, which is a formal derivation of the rules of the logic, used by the proof checker to conclude the validity of the specification. Checking a proof script is a mechanical process, efficiently performed by a piece of software. However, producing the proof script might require much more work. Despite the evolution of automated theorem provers, which apply sophisticated proof search algorithms, complex proofs—such as semantic preservation, and others, based on program transformations—are out of the reach of these tools. *Interactive proof assistants*, where the user constructs the proof while being guided by the software (which verifies the script and proposes some basic automation), become necessary in such cases. In this thesis, we use the Coq [21] proof assistant to perform the mechanical verification of our proofs.

2.1.1 Coq

Safety-critical systems require high levels of assurance in the compliance with the specifications, which must themselves be precisely stated in a formal language. *Proof management frameworks*, such as Coq, allow both the specification of algorithms, as well as the automatic and safe extraction of these specifications into executable programs.

Coq is also defined as an *interactive proof assistant*. A few other examples of proof assistants are Isabelle/HOL [57] and PVS [58]. They all have in common the fact that algorithms and theories are expressed in a single formalism, which allows the definition of functions, relations, theorems and properties which are interrelated. For instance, one can define a *sorted* relation on lists, then a *sort* function which computes sorted lists, and then a theorem which proves that all lists produced by *sort* are indeed sorted, according to the *sorted* relation. Many proof assistants also provide the ability to produce executable code from a given (executable) specification, in a mechanism called *code generation*. Figure 2.1 presents an overview of this process.

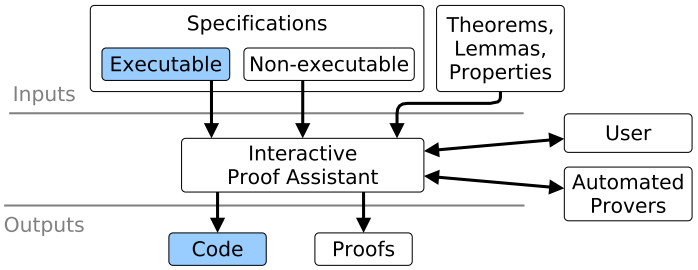


Figure 2.1: Overview of the architecture of an interactive proof assistant. Specifications and theorem definitions are inputs, some of which entail the creation of proofs. These proofs, built with the help of user interaction, are the reason for using the proof assistant. In some cases, it is also useful to obtain executable code produced by the proof assistant. This code is *proved* to correspond to a given specification (the correctness theorem).

The user first defines a specification, which can be executable (functional) or non-executable (relational). While a relational specification is often more natural (for instance, to express non-determinism), if automatically-generated executable code is needed, then the specification must be defined in a functional form. Afterwards, the user can define theorems and properties related to the specification, for instance, that a given computation produces a value which respects the specification. In some cases, the user may want to prove a given property about a piece of untrusted code. The use of *a posteriori validation* (explained in Section 2.1.2) allows this to be performed using the proof assistant. In this case, the specification will define a *validator* that will check the result of the untrusted code.

When a theorem is defined, the proof assistant switches from a *specification mode* into a *proof mode*, where it outputs the current goal state, and the user responds with the application of rules, theorems and transformations which allow the goal state to advance, until it is proved. The proof assistant checks that each rule application is logically valid, until the goal is solved and the proof is completed. This process is repeated for each lemma and theorem. In the end, the proof assistant guarantees the validity of the specification. For executable specifications, some proof assistants offer the option to transform them into executable code[1] (e.g. Coq can produce OCaml, Haskell and Scheme code; Isabelle/HOL can produce Standard ML code; and PVS can produce Common Lisp code). The production of code this way avoids human translation and, in some frameworks, is guaranteed to respect the behavior of the specification.

Coq's specification language, Gallina, is used both for programming algorithms and to specify properties. From a programmer's point of view, Gallina acts as a functional programming language with sophisticated features that can be compiled into executable code. From a critical systems designer's point of view, Gallina acts as a rich specification language to model system behavior and its safety properties. Finally, from the point of view of a prover, Coq provides help in proof construction and a proof checker that can validate the correctness proof of the system. This integrated approach requires trusting only a minimal code base and on the code generation mechanism, limiting the possibility of bugs creeping into the system and endangering its integrity. The Coq tool has been used in all of our formalizations, both to program the analyzers and algorithms (when they are not validated *a posteriori*), and also to specify and prove their correctness.

2.1.2 *A Posteriori* Validation

A posteriori validation consists in checking some properties on the *result* of some computation, abstracting away how it has been obtained. That is, instead of proving once, *a priori*, for all possible results, we wait until a result is produced, and then *check* only this result, to see if it indeed has the desired property.

The main advantage of *a posteriori* validation is that it avoids the need of specifying and proving

[1] Code generation is usually performed using functional languages, because they match more closely the semantics of the specification. There are some developments [44] with imperative languages such as Java, but they rely on provers to verify the generated code.

some complicated computations. Usually, when proving a piece of code, we work on two objects of different natures: a property which we want to state or prove, and an algorithm whose result verifies the property. We then show that the algorithm, by construction, produces results which always verify the property. This can be quite complicated, however, especially if the construction procedure involves several *heuristics* and *ad hoc* reasonings which involve implementation issues (e.g. efficiency, optimality) that are unrelated to correctness. For instance, in the register allocation problem, which consists in mapping an unbounded number of program variables into a finite number of machine registers while minimizing the number of stack spills (which happen when no fresh registers are available), one needs to find a k-coloring for the interference graph. A k-coloring is an assignment of one among k different colors to each vertex of a graph, such that no two adjacent vertices receive the same color. The interference graph models conflicts between program variables: whenever two variables are both *live* at the same time, that is, their values are useful and cannot be overwritten, the interference graph contains an edge linking them, to indicate that they cannot receive the same color (machine register) during register allocation. In this problem, it is much harder to *compute* a solution (especially an *optimal* one) than it is to check if a given solution, an assignment of vertices to colors, is correct (i.e. no adjacent vertices have the same color).

Another advantage of *a posteriori* validation is proof maintainability: implementation details and heuristics may change without impacting the proof of correctness. In a development that is entirely proved *a priori*, changing any details in the implementation usually requires updating proofs. When using *a posteriori* validation instead, many internal differences can be introduced without affecting the validator (e.g. in the register allocation example, the coloring heuristics can be modified).

A drawback of such validation is the need to perform *runtime checking* of properties: instead of having a proof that is valid for any execution of the function, we have to perform a validation during the execution of the program to be validated, for each result produced by the untrusted algorithm. The best cases are those where such validation is cheap, such as the test of valid coloring for register allocation. A validator for properties which are expensive to test (e.g. optimality of a shortest-path algorithm) might incur a significant performance loss.

The validator in Figure 2.2 illustrates its integration in the proof assistant: we combine an untrusted piece of code, which generates candidate results, with a specification of the properties to validate. This specification, defined in an executable way, checks that the candidate solution complies with all desired properties, otherwise validation fails. The correctness theorems using the validator must include the hypothesis that the validator succeeded (hence the *partial* correctness). It is possible to create a validator which has been proved correct, but fails too often to be actually useful. For instance, if a desired property has been incorrectly specified, it can result in a condition that is stronger than necessary and that, in practice, is never met. For this reason, some *testing*, via an experimental evaluation of the executable code, may be necessary to check that the generated code for the validator succeeds with typical input.

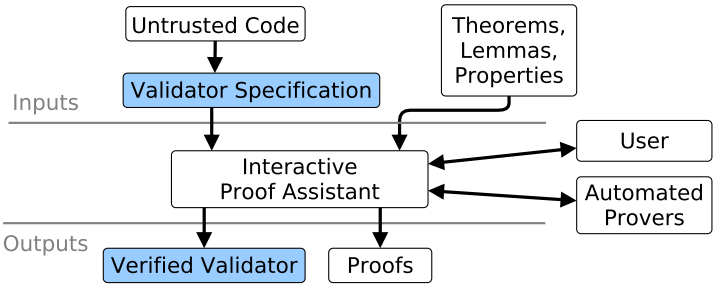


Figure 2.2: Overview of the *a posteriori* validation mechanism used with Coq. The general structure is very similar to the one used in *a priori* proofs (Figure 2.1).

We apply this technique in several parts of our development: loop reconstruction (Chapter 4), program slicing (Section 5.3.1), value analysis (Section 6.2.5) and IPET (Section 7.1).

2.1.3 Trusted Computing Base

A formal software development is comprised of several parts, specified and implemented by different people and using different methods. Each development invariably *trusts*, that is, relies without verifying, in a set of assumptions and subsystems; this set is called the *trusted computing base* (TCB).

The TCB takes into account the difference between *trusted specifications* and *verified components*, that is, which parts have been manually specified or designed and which parts have been mechanically verified. For instance, automated theorem provers—specialized software procedures capable of automatically deciding proofs in some logic fragments, such as first-order predicate calculus—are used to check the validity of proofs with respect to their specifications. Success means logical soundness, *supposing the prover itself is absent of errors* (due to the complexity of the highly-optimized algorithms implemented in automated theorem provers, errors are more frequent than would be expected). For this reason, the prover is part of the TCB, that is, trusting its output means trusting its implementation. If, on the other hand, the result of the prover is merely a *certificate* that can be checked by other, independent components, then the prover is considered as outside of the trusted computing base: an error in its implementation will, in the worst case, lead to an invalid certificate that will be rejected by the checker.

Formally verified systems strive to minimize the TCB whenever possible, since a smaller base implies a lesser chance of implementation errors, as well as a better traceability of the development. To do so, a widely used technique is to develop a minimal trusted *kernel* and then to extend it with components whose results can be verified by the kernel.

Coq's Trusted Computing Base In our developments, we rely on Coq to check our proofs, therefore our TCB includes Coq's TCB, that is, the proof-checking kernel and the code generation mechanism which produces the final OCaml code of our tools.

Coq's architecture satisfies the *de Bruijn criterion* [69], which states that *correctness must be guaranteed by a small checker*. That is, even though Coq is a large development[2], since its correctness only relies on a sufficiently small kernel (less than 10 kLoC of OCaml code), its TCB is eventually amenable to manual inspection. In practical terms, high-level Coq proof statements do not need to be checked individually, because when a proof is finished (using the `Qed` command), the kernel checker is run to ensure the constructed proof term is valid.

Concerning the code generation mechanism, it has been formalized on paper [48], and recent work [33] on its formal verification (using Coq itself) enables a high degree of trust in its correctness. Ongoing work should allow this verification to be concluded, thus removing this component from the TCB of every formal development which uses the code generation mechanism.

Finally, it is worth noting that CompCert and our development do not use some of the most sophisticated constructs in Coq, such as complex dependent typing features, to maintain the code relatively simple and to avoid unnecessary dependencies on all parts of the Coq development. Also, using code generation to different languages, it is possible to obtain several implementations of the same specification and compare them to check that their results are equivalent. All of these questions concerning trust are currently being studied by Coq and CompCert developers, to improve on the formal guarantees provided by the systems.

2.2 Overview of CompCert

The CompCert C compiler [10, 45, 46] is a machine-checked, optimizing C compiler. It can compile a large subset of ISO C99 code into several architectures (PowerPC, ARM and x86). Its distinguishing feature is that it is equipped with a formal semantics of the C and of the assembly languages, and also a *proof of semantic preservation* of the compiled code. Informally, this proof states that *the compiled code behaves as the source code*. In other words, no bugs are introduced during compilation.

We describe in Section 2.2.1 the overall architecture of the CompCert compiler, from the source down to the assembly level. Then, in Section 2.2.2, we present the formal definition of the semantic

[2]The Coq 8.4 release, including the standard library, consists in about 170 kLoC of Coq code and proofs, and 140 kLoC of OCaml code.

preservation of the compiled code, which we call CompCert's *correctness theorem*. In Section 2.2.3, we detail the annotation mechanism, which is used for our formally verified loop bound estimation, described in Chapter 4. Section 2.2.4 presents the main intermediate languages used in our analyses, RTL and Cfg. The following sections (Section 2.2.5 until Section 2.2.7) discuss the main aspects of this language related to our WCET-related static analyses, in particular the loop bound estimation.

2.2.1 CompCert Architecture

CompCert's highest-level language, CompCert C [47], follows closely the ISO C99 standard. Most features of C99 are implemented, and also a few C11 extensions are supported. There are several intermediate languages, from CompCert C down to assembly (which can be PowerPC, x86 or ARM). Not all the languages are detailed here, since most of them are not relevant for our analyses. Figure 2.3 presents the main languages which are used in our analyses: Clight and Cminor (two C-like languages, as expressive as C but with simpler syntax), used for some loop transformations described in Section 7.1, but mostly the RTL and Cfg languages, described in Section 2.2.4, the main languages used in our analyses.

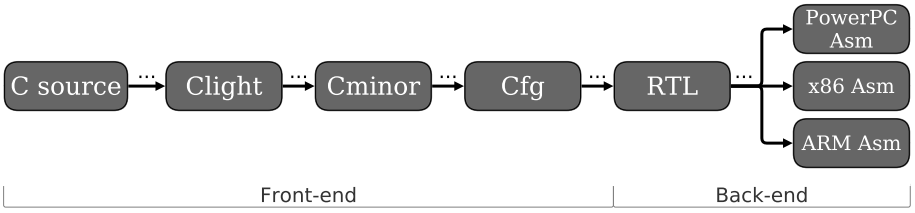


Figure 2.3: Main languages of the CompCert C compiler that are used by our analyses. The ellipses indicate the omission of transformations and other languages between those in the figure.

During compilation, the C source code is transformed multiple times, and eventually optimized, until the final assembly code is produced. At each intermediate language, a formal semantics of the language is specified and there are mechanically-verified proofs (in Coq) that the semantics is preserved. All of these proofs are compositional: the final correctness theorem chains them together and produces a result which is independent of the intermediate languages. In other words, the addition of intermediate languages is a convenience for the proof (since a succession of simple transformations is easier to prove than a single complex transformation) and it does not affect the trusted computing base. To minimize redundancy and to simplify the presentation, in this thesis we will present only the semantics of the Cfg language, and use it when describing the loop bound analysis in Chapter 4.

2.2.2 CompCert's Correctness Theorem

To express CompCert's correctness theorem, we must first mention which are the possible *behaviors* for programs, and then detail the concept of *observable event trace*. The correctness theorem is associated to the preservation of such behaviors and traces.

Program behaviors

In CompCert, programs have several possible outcomes. Applying semantic rules to a program gives a program *behavior*. There are three kinds of behaviors:

- *termination*: reaching a final state (i.e. exiting the `main` function) in a finite number of execution steps;
- *divergence*: executing an infinite number of steps without ever reaching a final state;
- *going wrong*: reaching a *stuck* state, from which no semantic rule can be applied.

Termination is the usual, expected behavior for the kind of embedded programs we consider here[3]. Divergent programs are common in many settings, and also have a defined semantics. For both termination and divergence, the program behavior also includes a (possibly infinite) trace of input/output operations. Programs that *go wrong* (for instance, C programs with out-of-bounds array accesses) are not defined by the language semantics.

We denote by $Behaviors(P)$ the set of all possible behaviors[4] of program P, $P \Downarrow B$ the execution of program P with behavior B (that is, $B \in Behaviors(P)$), and $Safe(P)$ the fact that P is a safe program ($P \Downarrow B \implies B$ *does not go wrong*).

Theorem 1 (Correctness of Compilation)**.**
Let S be a source program and C the result of its compilation.
If $Safe(S)$, then $C \Downarrow B \Longrightarrow B \in Behaviors(S)$.

This correctness theorem implies the preservation of functional specifications defined on the source code. For instance, when considering memory safety, Theorem 1 implies that, if S does not go wrong, neither does C: $Safe(S) \Longrightarrow Safe(C)$. This notion of correctness also means that a program that terminates cannot be compiled into a non-terminating program, and vice-versa.

Observable events trace

An important component of the behavior of programs is the trace of events it generates. The execution of a program is comprised of two parts, observable events on one side, and *internal computations* (also called *silent events*) on the other. Distinguishing between them is important for correctness and optimization. Observable events, as defined in the semantics, are situations which can be observed by the external world, mainly input/output operations and system calls. These events cannot be rearranged or optimized away without changing the semantics of the program. There are three kinds of observable events, and each registers some specific information about the event:

- system calls, with the associated call name, parameters and result;
- loads and stores of `volatile` global memory locations, with the associated memory address and data;
- annotations (described in Section 2.2.3), with the associated arguments.

Internal computations are steps which cannot be observed by the external world, typically intermediate computations and temporary values which are not exposed (printing a variable, for instance, exposes it). Internal computations can be optimized away, reordered or replaced, as long as they do not affect the values of observable events.

The preservation of observable behaviors by the compiler implies the equality of the traces of both source and compiled programs. This property is used, for instance, to prove the correctness of the loop bound estimation in Chapter 4.

2.2.3 Annotation Mechanism

CompCert provides an *annotation mechanism*, which enables some information to be transported from the source down to the assembly code. This mechanism serves different purposes: it allows precise tracking of program points between the source and the compiled code (for instance, to track loop entries and exits), and also enables mapping C source variables to their corresponding assembly locations (for instance, which machine register contains the variable value). Typical applications include debugging and assembly code inspection.

[3]The programs we consider here are *tasks*, typically invoked by a scheduler. The scheduler itself, not considered here, usually contains an infinite (divergent) loop.

[4]In practice, most C programs are deterministic and have a single possible behavior, but the C standard defines the language as non-deterministic (e.g. in `x = f() + g()`, either `f()` or `g()` may be evaluated first). Even if the formal C semantics must consider all behaviors, compilers typically choose one evaluation order, which leads to deterministic programs.

CompCert annotations are inserted in the source by calling a special built-in function (labeled `_annot` in our examples), with optional parameters. During program execution, calling this function generates an observable event, just like system calls, but in the assembly code there is no actual executable code, just a comment representing the program point where the annotation was inserted. Annotation arguments are parsed and transported through compilation. They can be either strings or program variables. Similarly to the `printf` function, arguments of the form `%1`, `%2`, etc., are placeholders for variables. Unlike `printf`, however, they are not replaced during runtime by the variable *value*, but instead they are replaced during compilation by the compiled variable *name*. For example, Figure 2.4 presents a small C code fragment containing a loop and the corresponding assembly code. The C source file contains three annotations: before, inside and after the loop. The annotation inside the loop contains a special argument, a reference to variable i. During compilation, this annotation is transformed to include a reference to the actual machine register corresponding to this particular occurrence of i (register `%ebx` in Figure 2.4).

The annotations themselves correspond to character strings without an associated semantics. However, due to the fact that they generate events in CompCert's observable trace, they cannot be optimized away and their relative order in the trace is the same in both original and compiled programs. Therefore, we know that the number of events generated in the trace corresponds exactly to the number of times the program executes the program point with the annotation. This property is used in the correctness theorem of the loop bound estimation (Section 4.4).

```
_annot("before␣loop");                  # annotation: before loop
i = 0;                                    xorl %ebx, %ebx
while (i < 5) {                         .L100:
  _annot("i is register %1", i);        # annotation: i is register %ebx
  i++;                                    leal 1(%ebx), %ebx
                                          cmpl $5, %ebx
}                                         jl .L100
_annot("after␣loop");                   # annotation: after loop
```

Figure 2.4: The annotation mechanism of CompCert allows the mapping of source code variables to machine registers. It also enables the correlation of program points between the two languages.

2.2.4 The Cfg and RTL Intermediate Languages

CompCert has many intermediate languages, each of them tailored for a specific purpose. In this thesis, part of the analyses have been developed in the CompCert Cfg language, while others have been developed in RTL (for *Register Transfer Language*), which is very similar to Cfg. We briefly present here both languages and the relevant differences between them. Since the languages are very similar, to simplify the presentation and avoid alternating between them in the manuscript, we decided to use the Cfg language as reference, even when the actual Coq implementation has been defined in RTL. They are very similar, except that Cfg has structured expressions and is architecture-independent, unlike RTL.

The CompCert Cfg Language

In the CompCert Cfg language, each function is represented by its control flow graph. Figure 2.5 contains a small C program (whose results are nonsensical) on the left and the compiled Cfg code on the right. It illustrates some elements of the Cfg language, for instance, the fact that each instruction has an associated program point $l \in \mathcal{PP}$. The program includes function calls (`f` and `scanf`), memory accesses (local array `res[i]`, global variable `g`), control flow structures and arithmetic operators.

Cfg has the same expressive power as C, but it is simpler due to transformations performed by previous compilation passes. For instance, expressions are free of side-effects, and array variables are mapped into memory, used via load/store instructions. Cfg being an architecture-independent

```
#include <stdio.h>
char g; /* global */

int f(int x) {
  if (x <= 1) {
    return 1;
  } else {
    return x * f(x - 1);
  }
}

int main() {
  int res[4];
  res[0] = g;
  int i = 1;
  int n;
  scanf("%d", &n);
  while (i < n) {
    res[i] = res[i-1] + f(i);
    i++;
  }
  return res[3];
}
```

```
f(x) {
  local vars: [x, $26]
      1: if (x <= 1) goto 2 else goto 3
      2: return 1
      3: $26 = call "f" (x - 1)
      4: return x * $26
}

main() {
  local vars: [i, $27]
      1: int32[&8 + 4 * 0] = int8signed["g"]
      2: i = 1
      3: call "scanf$ii" ("%d", &0)
      4: skip
      5: if (i < int32[&0]) goto 6 else goto 10
      6: $27 = call "f" (i)
      7: int32[&8 + 4 * i] = int32[&8 + 4 * (i - 1)] + $27
      8: i = i + 1
      9: goto 4
     10: return int32[&8 + 4 * 3]
}
```

Figure 2.5: Example of a C program (left) compiled into Cfg (right).

language, analyses performed at this level can be compiled into any assembly language supported by CompCert. In C, there are several kinds of loop structures (`for`, `while` and `do-while`) and control flow constructs (`break`, `continue` and `goto`), while in Cfg control flow is encoded explicitly: every Cfg instruction knows its direct successors.

The abstract syntax of the CompCert Cfg language is presented in Appendix A. It comprises several types of constants: integers, floating-point numbers, and memory addresses—which can be references to variables and function names, or to elements in the stack (e.g. array variables defined locally in a function). Cfg contains structured expressions, such as `int32[&8 + (4 * (i - 1))]`, which include most standard C arithmetic and logical operators, but also memory loads (e.g. the `int32` part in the expression represents the addressing mode of a load operation, here a signed 32-bit integer memory chunk). Finally, Cfg instructions include assignments, conditional branches, function calls and memory stores. Each instruction has labels indicating its successors.

Cfg variables are divided in three types: global variables, which are always stored in memory (more precisely, in the *heap*); and two types of local variables (a distinction which does not exist in C): *Cfg stack variables* and *Cfg local variables*. The former include variables whose address is taken (e.g. via the *address-of* (`&`) operator), such as arrays, while the latter comprise the remaining local variables, which are stored in an unbounded set of machine registers[5]. The rationale for having these two kinds of variables is that, by explicitly mapping some C local variables to machine registers, CompCert minimizes the number of variables in memory, which simplifies and improves the precision of several analyses and optimizations. For instance, variables mapped to registers do not suffer from aliasing issues.

Figure 2.5 displays occurrences of each kind of Cfg variable: one global variable (`g`), several local variables (`x`, `i`, `$26` and `$27`), and two stack variables (`n` and `res`, respectively `&8` and `&0` in the Cfg code).

[5]Register allocation, performed later in the compilation chain, will map them to the actual machine registers, or spill them into the stack if necessary.

A Cfg program is a triplet $(fs, id_{\text{main}}, gvars)$ containing a set of function definitions *fs* including a `main` function, its identifier id_{main}, and a set of global variable definitions *gvars*. Each function contains the set of its parameters (seen as local variables), its code (a mapping from program points to instructions) and the program point of the (unique) entry point. *External* functions (e.g. built-ins and library functions) can also be called (their source code is not available to CompCert, but their binary code can be linked after compilation). Some built-ins, such as `malloc` and `memcpy`, have their semantics specified in CompCert.

The Cfg language is not part of the standard CompCert distribution. It has been developed by members of the Verasco ANR research project as part of an effort to produce a verified static analyzer for C. It has been designed as a higher-level version of RTL (presented in the following) to compensate for some of its shortcomings in relation to static analyses.

RTL

Historically, the language of choice for program optimizations in CompCert is RTL. RTL is similar to an extended 3-address code[6] with explicit control flow information that is suited for data-flow analyses and optimizations, such as constant propagation, common subexpression elimination and tail call elimination.

RTL is an architecture-dependent language, although most of its code is shared between multiple architectures. Many of its operators are common between several architectures, but not all of them, therefore some analyses need to handle each architecture differently. Also, by virtue of being a 3-address-style language, RTL has no structured expressions. These are split into several instructions when compiling to RTL, which introduces a potential loss of precision for some analyses (an example is presented in the value analysis described in Chapter 6). To remediate this situation, the Cfg language has been introduced with the purpose of enabling more powerful static analyses within CompCert, while keeping several common aspects with RTL. Both languages have the same control flow structures, but Cfg has structured expressions. For instance, where Cfg keeps a structured expression such as $(1 + 2) \times 3$ in the form of a tree, RTL breaks it down into two instructions, one computing $1 + 2$ and storing it in a temporary t, and the other computing $t \times 3$. This has some consequences for the static analyses performed in RTL:

- RTL programs have more instructions, and many more variables, due to the creation of temporaries for intermediate results; therefore analyses in RTL need to be more scalable in order to deal with large programs;
- a non-relational value analysis performed in RTL is less precise than one in Cfg, since without structured expressions, some information between variables is lost.

The following example of a C program, compiled to Cfg and then to RTL, illustrates some distinctions between these two languages:

```
/* C source */          /* Cfg program */              /* RTL program */

int main() {            main() {                       main() {
  int i, j;               local vars: [i, j]             local vars: [x1, x2, x3, x4]
  i = 0;                  1: i = 0                       1: x1 = 0
                                                         2: x2 = x1
  j = 3 * i;              2: j = 3 * i                   3: x3 = x2 + x2 * 2 + 0
                          3: skip                        4: nop
  while (i < 5) {         4: if (i < 5) goto 5           5: if (x1 <s 5) goto 6
                             else goto 7                    else goto 8
    j += i;               5: j = j + i                   6: x3 = x3 + x1 + 0
  }                       6: goto 4                      7: goto 5
                                                         8: x4 = x3
  return j;               7: return j                    9: return x4
}                       }                              }
```

[6] RTL instructions may contain more than three operands, since they are related to the instructions of the underlying machine architecture, some of which (such as *add with shift plus offset*) contain several operands.

We notice that the main difference between the C source and the Cfg code in this program is that the control structure has been modified: loops become sequences of `if`s/`goto`s. However, expressions are left mostly untouched. Then, between Cfg and RTL, the control structure is the same, but each composite expression has been split into several operations. Variables i and j are replaced with pseudo-registers `x1`, `x2`, etc. A slightly larger example program, compiled from C to Cfg and then to RTL, is presented in Appendix B.

When considering most of the analyses performed in this thesis, namely loop transformations, loop reconstruction and program slicing, there is little difference between these two languages, since the control structures are identical. The main difference is seen in the value analysis, due to structured expressions. For historical reasons, most analyses have been implemented in RTL, however to improve readability and to avoid switching between two languages, in this manuscript we present all analyses as having been performed in the Cfg language. We indicate whenever there are issues concerning the differences between the languages.

In the rest of this section, we present definitions and properties related to the Cfg language that will be useful for our analyses and proofs. These notions will be referred to throughout the text.

Program Variables

The domain of program variables we consider in our analyses, defined as $\mathcal{V}ar \triangleq \{Reg(id) | id \in \mathbb{N}\} \cup \{Mem\}$, classifies Cfg variables into one of two kinds: either as a *pseudo-register* $Reg(id)$, where id is a unique identifier, or as a variable stored in memory, in which case it is represented simply by Mem. Mem represents the entire memory space, for analyses which do not distinguish between different memory locations. To do so requires some sort of pointer analysis, such as the one described in [64]. The term *pseudo-register* is used to distinguish program variables, whose number is unbounded, from the actual machine registers, limited in number, which are only known after register allocation is performed in the back-end.

Def/Use Functions

Using our definition of program variables, we define two functions, *def* and *use*, of type $\mathcal{PP} \to \mathcal{V}ar^*$, which map program points to sets of variables. These functions associate to each program point l a set of *possibly* modified (respectively referenced) variables for the instruction associated to l. They are obtained by syntactic inspection of the source code.

For instance, $def(\texttt{assign}(id, e, l)) = \{Reg(id)\}$, that is, the left-hand side of an assignment to id defines the value of the pseudo-register $Reg(id)$. Similarly, Mem is defined in any `store` assignment. An instruction containing a memory load expression implies that Mem is in the *use* set of this instruction. This set also contains all variables occurring in any expression in the instruction. Finally, the `call` instruction is dealt with in a conservative manner: since *def* and *use* are defined to work intraprocedurally (they do not track information across function calls), the *def* set of a `call` instruction includes not only the assigned variable id (if the function has a return value), but also the memory (Mem), because it may have been modified during execution of the called function.

Control Flow Graph

The vertices of our control flow graphs (CFGs[7]) are instructions, and program transitions form its edges. The CFG itself is a mapping from program points to instructions; each instruction is explictly associated to the program points of its successors.

Paths in a Control Flow Graph

A *path* in a CFG is a list of adjacent vertices. We note a path from l to l' through vertices $ls = [l, l_1, l_2, \ldots, l_n]$ as $l \xrightarrow{ls} l'$. Note that, to simplify some definitions, we include the initial vertex l in the list of vertices of the path, but not the final vertex l'. As an example, in Figure 2.6 there is a path $[2, 3]$ from 2 to 6. Note that a path is only completely specified when the start and end

[7] To distinguish between the Cfg CompCert language and the abbreviation for a control flow graph, the latter is written in capital letters.

vertices are specified, i.e. $[2, 3]$ in itself does not constitute a path. For every vertex, there is a trivial empty path from itself to itself. A *cycle* is a non-empty path from any vertex l to itself. For instance, the path $[6, 2, 3]$ constitutes a cycle from 6 to itself in Figure 2.8. Finally, for any vertex l', we say that it is *reachable* from a vertex l if there exists a path from l to l' (symmetrically, l *reaches* l').

Normalized Control Flow Graphs

In the CompCert Cfg language, the CFG of a program may contain (1) unreachable vertices, (2) arcs arriving at the entry point (if the entry is in a loop), and (3) multiple exits. Reasoning on such CFGs is unnecessarily complex, since they can always be transformed into semantically equivalent programs which do not have these properties, that is, programs having CFGs where:

- the entry point l_{entry} has no predecessors (i.e. no vertices have l_{entry} as their successor);
- all vertices are reachable from l_{entry};
- all vertices can reach l_{exit};
- there is a unique exit vertex l_{exit}.

Figure 2.6 depicts a CFG (with $l_{\text{entry}} = 1$ and $l_{\text{exit}} = 6$) where none of these properties are respected, followed by the same CFG after normalization (with $l_{\text{entry}} = \texttt{en}$ and $l_{\text{exit}} = \texttt{ex}$). By construction, most Cfg programs respect these four properties[8], but they need to be validated before they can be used in proofs. We perform these validations to ensure CFGs are normalized. In practice, all programs in our evaluation compile to normalized CFGs.

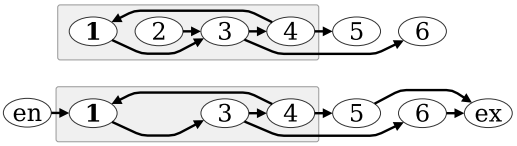


Figure 2.6: Non-normalized CFG (top) and its normalized version (bottom). Vertex 2, which is unreachable, has been removed, and extra vertices have been added: `en`, the unique entry, and `ex`, the unique exit.

2.2.5 Loops and Loop Nestings

Loops are control structures that are particularly challenging for compilation, static analysis and WCET estimation, for many reasons: they can incorporate infinite behavior, they are often responsible for most of the execution time of a program, and they can be very costly for static analysis (e.g. when trying to obtain a fixpoint). Loops come in many shapes, some more frequent than others, and in their most general form they require sophisticated reasoning.

In this document, we consider a restricted form of loops, sufficiently general so as to accept most useful programs, but at the same time having stronger structural properties which help our proofs. We consider *reducible* loops [53], that is, loops having a single entry vertex. Programs with irreducible loops are very rare, and there are techniques to convert irreducible loops into reducible ones, such as *node splitting* [2]. As examples of CFGs containing reducible and irreducible loops, Figure 2.7 depicts on the left a CFG with only reducible loops, while the CFG on the right contains an irreducible loop `(3 4)` (both 3 and 4 are loop entries, since they are both successors of 2, which is outside the loop).

We define a loop as a sequence of vertices which form a strongly connected component (i.e. there is a directed path from any vertex in the loop to any other vertex in the same loop). Loops can be *nested*, that is, they can occur inside other loops. Loop entry vertices (those having predecessors

[8] CompCert passes such as *Renumbering* eliminate unreachable vertices. By default, functions with multiple returns are compiled with a single exit vertex.

outside of the loop) are called *headers*. More precisely, the loops in our programs are represented by a structured record *Loop* containing the following fields:

- *vertices* : $\mathcal{PP}^+$, the list of program points that are inside the loop;
- *header* : $\mathcal{PP}$, the single entry vertex of the (reducible) loop;
- *parent* : *Loop*, the immediately enclosing loop (or the entire function[9], if the loop is not nested within another one).

The *parent* field of the *Loop* structure is related to *loop nesting forests* [60], that is, the nesting relation between loops in a program. This relation defines a set of trees where each child node is a loop nested within its parent. Our *Loop* structure can be extended to consider the whole CFG as a *base loop*, the parent of all first-level loops (such as (2 3 4 5 6) on the left CFG in Figure 2.7). Its header is the program entry (l_{entry}) and its parent is itself.

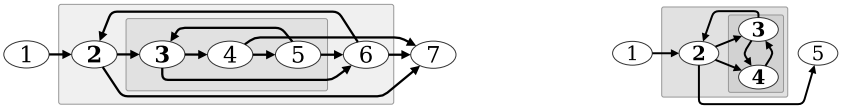


Figure 2.7: CFGs illustrating the different kinds of loops: the left one only has reducible loops, while the right one contains a reducible loop (2 3 4) nesting an irreducible loop (3 4). *Loop headers* (loop entry vertices) are written in bold.

We use Bourdoncle's algorithm [13] to reconstruct the loop structure of our Cfg programs. This algorithm is similar to Ramalingam's [60] method to compute loop nesting forests, and also to Lengauer and Tarjan's [43] algorithm for finding dominators. These algorithms all have in common the fact that they use several properties about graph theory and rely on imperative data structures for efficiency. Proving their correctness is a challenge in itself, since it requires formalizing and reasoning about structures such as *postdominator trees*. We opt instead for an axiomatization of the loop nesting structure and the use of *a posteriori* validation to enforce the properties necessary for our proof. This approach allows us to use an efficient algorithm for loop reconstruction, since this algorithm does not need to be proved. It produces a *Loop* record for each loop in the program, plus a function *loop*: $\mathcal{PP} \to$ *Loop* mapping each program point to the *smallest* loop containing it (the same program point may belong to several loops, e.g. vertex 3 in Figure 2.7 belongs to loops (2...6) and (3...5)).

The loop validator only has to verify, for each vertex and reconstructed loop, that the properties needed for the proof are satisfied. This is simpler than the actual reconstruction and can be done efficiently without resorting to complex data structures and algorithms. We describe informally some of the properties that need to be validated:

- *Single entry point*: there must be a single vertex in each loop with predecessors outside of the loop, i.e. each loop must be reducible.
- *Single loop entered*: CFG arcs can only enter one loop at a time. For instance, if we had a loop structure such as (1 ((2 3) 4) 5), the arc (1,2) would not be allowed, since it would enter both loops (2 3) and (2 3 4). A consequence of this property is that each loop has at least one proper vertex. A vertex is *proper to a loop* if it is not inside another more deeply nested loop. This avoids empty loops and simplifies the loop nesting structure.
- *Cycles must include headers*: any cycle from a vertex l to itself must include the header of l (that is, $\mathit{header}(\mathit{loop}(l))$). This property is a consequence of another one, namely that *all back edges point to a loop header*. Its corollary is that we can bound the number of iterations of any vertex in a loop just by bounding its header.

[9] We consider here an intraprocedural analysis, where each function is a separate CFG. As we will see later, our analysis will be extended to the entire program.

Among these properties, the first one is the only which can cause programs to be rejected; the two others are guaranteed by construction by the Bourdoncle algorithm. The validator performs a single traversal of the CFG, with the proposed loop structure and performs consistency checks (e.g. that every vertex is inside its own loop) which ensure all properties of the axiomatization. They are used mainly in the proof of correctness of the loop bound estimation (Section 4.4).

2.2.6 Simplifying the Cfg Language for a WCET Analysis

In this section, we explain how we can take into account assumptions related to critical real-time systems programming to perform an interprocedural analysis for Cfg, improving the precision of the analysis. We call this modified intermediate language ICfg and use it to define the semantics that will be common to all analyses in this thesis.

Critical Real-Time Systems Assumptions

Real-time systems programming imposes some constraints on the general form of programs, such as:

- recursion is not allowed, or explicitly bounded;
- loops are explicitly bounded, or at least supposed to terminate;
- function pointers are not allowed, or they must be statically determined.

Some of these constraints are related to performance issues (e.g. in a language without recursion, function calls may use a statically allocated stack frame, saving some instructions during runtime), but in general these constraints are related to software issues. They improve the ability to statically allocate resources and to predict program behavior (e.g. to avoid memory overflow). For the programmer, these constraints help avoiding mistakes which might generate infinite loops or excessive memory consumption.

From the point of view of a formal development, these constraints are useful for several static analyses: our loop bound and WCET estimations, including its components such as program slicing, make use of these hypotheses. Related but different subjects, such as automatically proving the termination of programs (e.g. via *ranking functions* [3]), are not considered in this thesis.

Inlining for an Interprocedural Analysis

Interprocedural analyses, which are often extensions of intraprocedural analyses with information about call contexts, are precise but often costly, especially when flow-sensitive. In the real-time systems community, where restrictions such as those previously mentioned are commonplace, an alternative approach can be much more interesting from a cost/benefit point of view: the elimination of function calls through the use of *function inlining*.

Function inlining is a classical transformation in compiler literature and in static analysis which replaces the instruction corresponding to a function *call* with the entire code of the function being called. Variants include *virtual* inlining, where the actual program code is not modified, even though the analysis inspects the code of each called function, and *physical* inlining, where the program is modified: each function call is replaced with a copy of the code of the called function. Inlining is useful to avoid overhead related to function calls (such as the allocation of a new stack frame, copy of effective parameters and jump/return), at the cost of a possible increase in code size (for multiple calls of the same function). This transformation is especially useful when short functions are inlined, but inefficient and memory-consuming (due to code size explosion) in the presence of repeated calls to large functions. Note that inlining is not an ideal solution for industrial-scale analyses dealing with large code bases, where it is usually better to resort to an interprocedural approach. Otherwise, selective (partial) inlining is also able to maintain scalability, at the cost of reduced precision.

In a program without recursion, physical inlining is able to eliminate all function calls. In this case, *the interprocedural semantics of the whole program execution becomes equivalent to the intraprocedural semantics of the* `main` *function call*. This equivalence is important when reasoning about Cfg programs in our analyses. In the WCET-related analyses and benchmarks, function

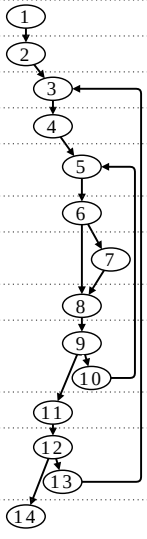
C Source Code (*main* function)	*n*	Cfg Program (*main* function)	Control-Flow Graph	succs(*n*)	*def*(*n*)	*use*(*n*)
`int a[6], i, j;`						
`i = 0;`	1	`i = 0`		{2}	{i}	∅
`do {`	2	`skip (goto 3)`		{3}	∅	∅
`  j = 0;`	3	`j = 0`		{4}	{j}	∅
`  do {`	4	`skip (goto 5)`		{5}	∅	∅
`    a[i] += i + j;`	5	`int32["a"+4*i] = int32["a"+4*i] + (i+j)`		{6}	{*Mem*}	{*Mem*,i,j}
`    if (i==j)`	6	`if (i == j) goto 7 else goto 8`		{7,8}	∅	{i,j}
`      a[i] *= 2;`	7	`int32["a"+4*i] = int32["a"+4*i] * 2`		{8}	{*Mem*}	{*Mem*,i}
`    j++;`	8	`j = j + 1`		{9}	{j}	{j}
`  } while (j≤5);`	9	`if (j <= 5) goto 10 else goto 11`		{10,11}	∅	{j}
	10	`skip (goto 5)`		{5}	∅	∅
`  i++;`	11	`i = i + 1`		{12}	{i}	{i}
`} while (i≤5);`	12	`if (i <= 5) goto 13 else goto 14`		{13,14}	∅	{i}
	13	`skip (goto 3)`		{3}	∅	∅
`return;`	14	`return`		∅	∅	∅

Figure 2.8: Example C program (reduced to its *main* function) with the compiled Cfg code (which coincides with its ICfg code), and its control flow graph. The last three columns display successors per statement and def/use sets.

inlining enables context sensitivity and improved precision without incurring in significant costs in terms of analysis time—it is sufficient to check that the program does not contain recursive function calls. We present in the following the syntax and semantics of this *Inlined Cfg* (ICfg) language.

2.2.7 ICfg Syntax and Semantics

We define Inlined Cfg as an intermediate language derived from the CompCert Cfg language. Its syntax is identical to the one presented in Figure A.1 except for the *call* instruction, which does not exist anymore in ICfg. The *return* instruction terminates program execution. Figure 2.8 presents an example C program with the corresponding Cfg code and its control flow graph. Since this program has no function calls, the Cfg and ICfg code are the same. The program includes two nested loops, some array operations (with loads/stores), basic arithmetic and variable assignment. The Cfg/ICfg code is presented in a human-friendly syntax[10] equivalent to the one presented in Figure A.1.

The control flow graph of an ICfg program consists in a quadruple $(L, \texttt{succs}, l_{\text{entry}}, l_{\text{exit}})$ where L is a set of *program points* (identified by positive numbers), `succs` is a mapping from program points to sets of program points (their immediate successors in the CFG), l_{entry} is the program's entry point, and l_{exit} is the (unique) exit point (that is, a `return` statement). L defines the vertices of the CFG, while `succs` defines its arcs (as an adjacency list).

Semantic States

Figure 2.9 presents the small-step semantics of ICfg instructions. A state in this semantics is a triple (l, R, M) consisting of a program point l, a mapping R from pseudo-registers to values, and a memory state M, which can be seen as a map from addresses to values. The values associated to pseudo-registers are those from the CompCert compiler, namely:

- *Vint(i)*: a 32-bit integer i;

[10] For instance, the instruction associated to program point 7 is `store(int32,` $\texttt{addrsymbol}(a,0) + 4 * i$`,load(int32,` $(\texttt{addrsymbol}(a,0) + 4 * i) * 2), 8)$, but we write it as `int32["a"+4*i] = int32["a"+4*i] * 2`. The `goto 8` part is implicit since 8 is the syntactical successor of 7.

- *Vfloat(f)*: a floating-point[11] number f;
- *Vptr(b,o)*: a pointer value (consisting of the address of a block b and an offset in this block o);
- *Vundef*: an uninitialized (or unknown) value.

We will often regroup (R, M) as an environment E, when abstracting details related to the memory. To simplify the presentation of the ICfg semantics, in Figure 2.9 we note $P.\texttt{code}$ to indicate the code of the `main` function of program P, which is the only function in the ICfg program.

Program states: $S ::= (l, R, M)$
Register states: $R ::= r \mapsto v$

$$\frac{P.\texttt{code}(l) = \lfloor \texttt{skip}(l') \rfloor}{(l, R, M) \to (l', R, M)}$$

$$\frac{P.\texttt{code}(l) = \lfloor \texttt{assign}(id, e, l') \rfloor \qquad \texttt{eval\_expr}(e, R, M) = \lfloor v \rfloor}{(l, R, M) \to (l', R[id \leftarrow v], M)}$$

$$\frac{\begin{array}{c} P.\texttt{code}(l) = \lfloor \texttt{store}(chunk, addr, src, l') \rfloor \\ \texttt{eval\_mode}(chunk, addr, R) = \lfloor Vptr(b, o) \rfloor \qquad \texttt{storev}(M, chunk, b, o, R(src)) = \lfloor M' \rfloor \end{array}}{(l, R, M) \to (l', R, M')}$$

$$\frac{P.\texttt{code}(l) = \lfloor \texttt{if}(e_{cond}, l_{\text{true}}, l_{\text{false}}) \rfloor \qquad \texttt{eval\_expr}(e, R, M) = \lfloor Vint(0) \rfloor}{(l, R, M) \to (l_{\text{false}}, R, M)}$$

$$\frac{P.\texttt{code}(l) = \lfloor \texttt{if}(e_{cond}, l_{\text{true}}, l_{\text{false}}) \rfloor \qquad \texttt{eval\_expr}(e, R, M) = \lfloor Vint(i) \rfloor \qquad i \neq 0}{(l, R, M) \to (l_{\text{true}}, R, M)}$$

Figure 2.9: Small-step semantics of ICfg instructions

For each semantic step (`skip`, assignment, memory store and conditional branching), a necessary precondition is that the current program point must have an instruction associated to it in the code of the main function ($P.\texttt{code}$). The notation $f(a) = \lfloor b \rfloor$, for $a \in A$ and $b \in B$, denotes that $a \in \text{dom}(f)$, that is, f is a partial function from A to B (denoted as $A \nrightarrow B$ or $A \to option\ B$), a is in the domain of f, and f maps a to b[12].

The semantics of a `skip` is simply a transfer of the current program point, without modifying anything else. An `assign` instruction evaluates the expression to be assigned (which might be an arithmetic or logical operation, or a memory load) and, if the evaluation succeeds, the returned value is associated to id, which must be a Cfg local variable (a pseudo-register). Expressions are evaluated in big-step style, via the function `eval_expr`, which may either succeed or fail (e.g. due to a division by zero, or an out-of-bounds array access), in which case the semantics is undefined. The evaluation of expressions is not detailed here, but it can be found in the Coq development[13]. A `store` is an assignment to a variable in the memory. `eval_mode` is a big-step evaluation of the address where the data will be written, and `storev` performs the actual memory update, returning a new memory state M'.

[11] Floating-point numbers in CompCert are single- and double-precision IEEE 754 numbers.
[12] $\lfloor v \rfloor$ corresponds to `Some v` in Coq/ML.
[13] Cfg expressions are the same as Cminor expressions, which are detailed in CompCert's Cminor semantics.

Conditional branching is performed according to the result of the expression representing the condition to be evaluated, just like in C: if the evaluation returns the zero integer value *Vint(0)*, the l_{false} branch is taken; if it returns an integer value other than zero, the l_{true} branch is taken.

Initial and Final States

An initial state in the ICfg semantics is a triple $(l_{\text{entry}}, R_{\text{undef}}, M_{\text{init}}) \in \Sigma_{\text{init}}$, where l_{entry} is the (unique) entry point of the program, R_{undef} is the default mapping for pseudo-registers—which associates any pseudo-register to an uninitialized value (*Vundef*)—and M_{init} is the initial memory state. Just like in C, global variables have their values initialized to 0 and non-static local variables are uninitialized. A final state is a state of the form (l_{exit}, R, M), where l_{exit} indicates the program has reached the unique exit vertex of the program.

Steps and Reachability

An execution step in the small-step ICfg semantics is noted $\rightarrow$. Its reflexive and transitive closure is noted $\rightarrow^*$. An execution is composed of an initial state $\sigma_{\text{init}} \in \Sigma_{\text{init}}$ and a sequence of execution steps. The states visited during execution define an *execution trace*. For a given program P, we note $\texttt{reach}(P)$ the set of *reachable states* of P, that is, the states σ such that $\sigma_{\text{init}} \rightarrow^* \sigma$. An execution is called *terminating* if its last state has reached l_{exit}. Otherwise, we either have a diverging (infinite) execution, or a program that *goes wrong*. Our tools and analyses do not concern these cases.

Conclusion

We presented in this chapter some general concepts about formal program verification, as well as the tools used in this thesis to perform such verification, namely the Coq proof assistant and the CompCert compiler. We described in detail the Cfg language, and more specifically on its ICfg variant, since it will be used throughout the thesis as the implementation language of our analyses and the reference semantics of our proofs. In the next chapter, we describe the complementary context of this thesis, focusing on the subject which is the target of our formalization: WCET estimation methods based on static analysis.

Chapter 3

Context: WCET Estimation and Static Analysis

The subject of this thesis is the formal verification of WCET estimation based on static analysis. While the previous chapter focused on the formal aspects of proofs and semantics, this chapter presents the state-of-the-art of WCET estimation methods, along with the static analysis methods designed for our WCET estimation tool, namely value analysis and loop bound estimation.

Section 3.1 concerns WCET estimation. It details each of the main components of typical WCET estimation techniques. Section 3.2 is related to a value analysis based on abstract interpretation. It introduces the former and defines several concepts related to the latter. Value analyses are used in several parts of this thesis. Finally, we present a survey of related work on WCET estimation and compilation (Section 3.3).

3.1 WCET Estimation Techniques

Static WCET estimation techniques are composed of three main parts [70], as depicted in Figure 3.1:

1. Control flow analysis: determines information about control flow, including extraction of the control flow graph, recursion and iteration bounds, relative execution frequencies of vertices (e.g. relative frequencies of conditional branches) and infeasible paths. This is often obtained via a value analysis and a loop bound estimation;

2. Processor-behavior analysis: incorporates information about hardware components which influence program timing, such as cache hierarchy, pipelines and branch predictors;

3. Estimate calculation: computes global bounds based on the results of the previous parts. The most common method to compute these bounds is based on the *implicit-path enumeration technique* (IPET), which consists in generating an integer linear program describing the possible execution flows of the program, and then using a solver to obtain the worst case. This technique offers a good trade-off between precision and efficiency.

Control flow analysis is often performed on the reconstructed CFG of the binary program, because this is typically the starting point of WCET estimation methods. In principle, it can also be done on a high-level language (e.g. on the C source code), but in this case the tool needs a way to communicate this information to the machine level, such as some form of code mapping or annotation mechanism.

Processor-behavior analysis is performed at the binary level, usually after compilation and linking, otherwise it will lack necessary information such as memory mapping, required to perform cache analysis. It is performed after control flow analysis because it can benefit from information computed by the latter, such as infeasible paths. It computes timing information (in clock cycles) for each basic block in the program, taking into account several hardware elements (caches, pipelines, etc.).

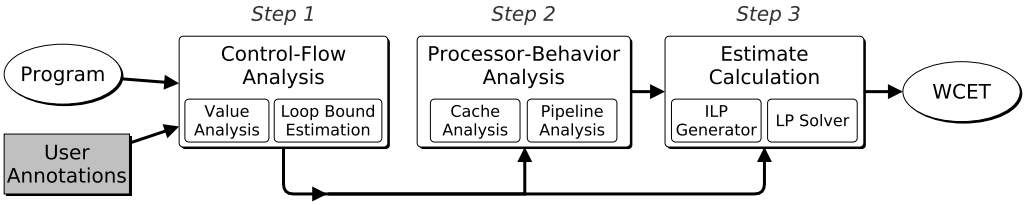


Figure 3.1: General architecture of WCET estimation tools. Starting from a program (and optionally a set of user annotations), they perform a control flow analysis, then a processor-behavior analysis, and finally an estimate calculation which outputs a WCET bound for the program. Each stage produces information used by its successors.

Estimate calculation uses the flow information obtained in step 1, plus the timing information from step 2, to define timing equations relating the different basic blocks in the program. They are then used to estimate the final WCET, usually via IPET. Since this step uses the result from 2, it is also performed on the fully linked executable.

In this thesis, we do not formalize the processor-behavior analysis. Its formalization requires modeling several architectural components, such as abstractions of the CPU and its memory hierarchy. Hardware models for real-life processors exist, but they are either not formalized (e.g. the ColdFire pipeline model by Langenbach et al. [42]) or not sufficiently precise with respect to timing elements such as cache and pipeline (such as the ARMv6 model proposed by Shi et al. [65]).

In this work, we focus instead on the control flow analysis, with the generation of flow annotations. We use a simplified cost model, a first step before incorporating a realistic one. In our model, instructions have a unitary cost, thus the WCET corresponds to the number of executed instructions along the worst-case path. Note that a new hardware timing model, obtained by modifying step 2, would also impact step 3, although to a lesser extent. For instance, the inclusion of cache and pipeline analyses requires the addition of new constraints (e.g. representing different pipeline states) to the IPET. In the end, for any given timing model, the solution of the IPET constraints, obtained via a linear programming solver, corresponds to the WCET estimation.

In this section, we start by presenting in detail each of the three main steps of a WCET estimation tool, then we conclude with a survey on tools relating WCET estimation and compilation.

3.1.1 Control Flow Analysis

The objective of a control flow analysis (CFA) in WCET estimation tools is to produce control flow information to be used by the following stages. This information is composed of data relating program points and program transitions, usually called *flow facts*. These facts contribute to obtaining timing bounds by constraining the execution count of instructions. Besides loop bounds, which are the typical information computed at this stage, flow facts may include upper bounds for branch executions (e.g. , an `if (i < 3)` inside a loop `for (i = 0; i < 10; i++)` will never execute more than 3 times per loop iteration) and similar information.

CFA often relies on the use of a value analysis to obtain bounds on program variables, which are then used to constrain control flow. Value analyses are more precise on the source level, because compilation reduces available information (e.g. by decomposing structured control flow and expressions), but performing a value analysis at the source level requires the correct translation of control flow information during compilation.

Flow Facts The flow facts we consider here are linear inequalities between CFG entities, that is, vertices and edges. Linear inequalities are sufficiently expressive for most common flow facts (loop bounds, relative frequencies, infeasible paths, etc.) and can be directly converted into constraints for an LP solver.

Figure 3.2 presents an example program with typical flow facts relating program points and program transitions. In the figure, vertices are numbered from 1 to 10 and edges are identified by their source and target vertices, except the start and end edges, named *en* and *ex*.

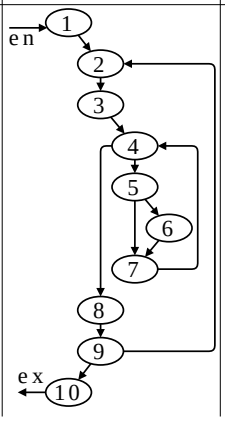
Program	CFG	Structural Flow Facts	Semantic Flow Facts	Execution Counters
`i = 0;`		$e_{en} = x_1 = 1$		1
`do {`		$x_2 = e_{1,2} + e_{9,2} = e_{2,3}$	$x_2 \leq 6$	6
`  j = 0;`		$x_3 = e_{2,3} = e_{3,4}$		6
`  while (j < i) {`		$x_4 = e_{3,4} + e_{7,4} = e_{4,5} + e_{4,8}$	$x_4 \leq 5x_3$	21
`    if (j % 2 == 0)`		$x_5 = e_{4,5} = e_{5,6} + e_{5,7}$		15
`      printf("a");`		$x_6 = e_{5,6} = e_{6,7}$		9
`    j++; }`		$x_7 = e_{5,7} + e_{6,7} = e_{7,4}$		15
`  i++;`		$x_8 = e_{4,8} = e_{8,9}$		6
`} while (i <= 5);`		$x_9 = e_{8,9} = e_{9,2} + e_{9,10}$		6
`return;`		$e_{ex} = x_{10} = e_{9,10} = 1$		1

Figure 3.2: Example program with its control flow graph, structural and semantic flow facts constraining the number of executions of each program point and transition, and the actual execution counters per program point.

Each CFG entity has an associated *execution counter*, defined as the exact number of times the entity is executed during the entire program execution. This information is dynamic; it cannot be exactly determined statically. We consider therefore static *approximations* of these counters, by defining two sets of variables, x_i associated to vertices and $e_{i,j}$ (or $e_{\text{en}}/e_{\text{ex}}$) associated to edges, which approximate the execution counters related to the respective CFG entities. Since we are interested in obtaining a safe upper bound for the WCET, we define these variables as *over*-approximations, and as valid for every program execution, including the worst case. A flow fact is a constraint relating such variables. Note that flow facts do not relate the execution counters themselves, only the variables representing their approximations.

Some constraints, called *structural*, are obtained directly from the control flow of the program. In their general form, they state that *the number of executions of a program point is equal to the number of executions of all entry edges, and both are equal to the number of executions of all exit edges.* This is the typical formulation of a flow network:

$$\sum_{p \in preds(i)} e_{p,i} \; = x_i = \sum_{s \in succs(i)} e_{i,s}$$

This results in the constraints displayed on the third column of Figure 3.2. For instance, $e_{2,3} = e_{1,2} + e_{9,2}$ is a structural constraint stating that we must leave vertex 2 (via 3) as many times as we enter it (via 1 or 9). Note that some execution constraints are redundant, for vertices having a single entry or exit edge.

Other constraints must be obtained through static analyses or must be given by the user. They are called *semantic* constraints, of which $x_2 \leq 6$ and $x_4 \leq 5x_3$ are two examples obtained via loop bound analysis. Here we only present loop bounds, but semantic constraints can relate all kinds of invariants between program points and transitions, to reduce the number of possible paths and improve the worst-case estimation. For instance, another valid semantic flow fact in this program would be $x_6 < x_5$, a strict inequality between these two vertices, since there is at least one case during program execution when the `if` condition is false.

Flow Fact Precision It is not always possible to obtain exact constraints if control flow is dependent on program variables. In Figure 3.3, we have an example of a *triangular loop nesting*, where the number of executions of the inner loop depends on a program variable whose value is different at each iteration. It can be hard (or even impossible) to statically predict the number of iterations of the instructions inside the inner loop. In the example in Figure 3.3, the best flow fact that most analyses can generate is $x_4 \leq 5 \times 5 \times x_2$ (where x_4 is inside the inner loop and x_2

is just before entering the outer loop). There are techniques to better deal with this, for instance using *relational analysis* to infer precise relations between i and j. In a relational analysis (such as a *polyhedral analysis*), instead of considering each variable in isolation, their domains are related, e.g. by linear inequalities. Precision comes at the cost of extra complexity and computation time.

```
1: int i, j;
2: for (i = 0; i < 5; i++) {
3:   for (j = 0; j <= i; j++) {
4:     ...
5:   }
6: }
```

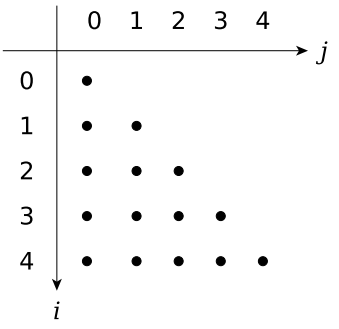


Figure 3.3: A typical example of a *triangular loop nesting*. It is called *triangular* due to the shape of the points formed by the values of the induction variables (i, j) during the iteration.

Other cases may be even more complex: the condition `j % 2 == 0` at program point 5 in Figure 3.2 is non-linear. One way to improve precision, at the cost of performance, is to split each constraint into several sub-constraints, each of them specific to one or several loop iterations. For instance, x_4 can be split as $x_{4_1} + x_{4_2} + x_{4_3} + x_{4_4} + \ldots$, where x_{4_i} corresponds to the number of executions of program point 4 during the i^{th} execution of the outer loop. We can then bound exactly each of these variables and obtain a more precise bound for program point 4. However, the two major problems with this approach are: (1) it is hard to infer automatically how many variables should be created, since the exact number depends on the number of iterations of the outer loop, and this information is not always known; (2) over-splitting can generate too many variables and equations, slowing down the resolution of the system. For instance, a simple loop with 1000 iterations would entail the creation of 1001 equations (one for each iteration, plus one for their sum). This approach is further discussed in Section 3.1.3, when describing IPET.

Facts for Loop Bounds Structural flow facts are sufficient only for programs without loops or recursion. Otherwise, assuming the worst case implies an infinite execution. Thus, semantic constraints are mandatory for loops and recursive calls. The simplest and most common annotations used are loop bound inequalities of the form $x_i \leq N$, where i is a vertex inside the loop and N is a positive constant. These constants define *global* bounds, that is: for a nested loop, or a loop defined inside a function other than `main`, the global bounds need to take into account the number of executions of *all* enclosing loops, plus possibly multiple function calls. For instance, let us consider the following program:

```
void f() {
  int k;
  for (k = 0; k < 4; k++) {
    /* loop3 */
  } }

void main() {
  int i, j;
  for (i = 0; i < 7; i++) {
    /* loop1 */
    for (j = 0; j < 5; j++) {
      /* loop2 */
    }
    f();
  } }
```

The first loop inside function `main`, denoted $loop_1$, is executed 7 times, that is, the transition from the loop condition (`i < 7`) to the loop body can only be taken 7 times. Supposing the first program point inside the loop body (statement `j = 0`) is numbered n, the semantic constraint $x_n \leq 7$ is valid. However, the same methodology cannot be applied to the other loops: if the first program point inside the loop body of $loop_2$ is n', then $x_{n'} \leq 5$ is *not* a valid constraint, but $x_{n'} \leq 35$ is. The same reasoning is necessary for $loop_3$: let n'' be the first program point inside the loop; then $x_{n''} \leq 28$ is a valid flow fact, but not $x_{n''} \leq 4$.

Since the actual values for loop bounds depend on their enclosing contexts (loops or functions), it is often easier to introduce *local bounds*, which are context-free, by establishing relative constraints between the counters of program points immediately before the loop, and those inside the loop body. In our previous example, if we consider m, m' and m'' as the program points just before the loop conditions (statements `i = 0`, `j = 0` and `k = 0`, respectively for loops 1, 2 and 3), then the following flow facts are valid: $x_n \leq 7 \times x_m$, $x_{n'} \leq 5 \times x_{m'}$ and $x_{n''} \leq 4 \times x_{m''}$. It is easier to generate such constraints, but proving them is not easier than proving the global bounds. In Chapter 4 we detail the correctness proof of global loop bounds, and in Chapter 7 we explain how to generate constraints from such bounds. The intuition is that the x_i variables are *global* counters, and therefore proving local properties about them requires as much (or more) effort as proving the global bounds. Besides, we show in Chapter 4 that we can always transform loops to ensure that the local bounds presented here can be replaced by equivalent global bounds.

Obtaining loop bounds is one of the primary concerns of the flow analysis stage. Since finding such bounds is undecidable in the general case, each WCET estimation method provides a different trade-off between precision of the estimated bounds, efficiency of the analysis, and percentage of loops actually bounded. In Section 4.1 we present the control flow analysis which served as basis for the method we formalize in this thesis.

3.1.2 Processor-Behavior Analysis

Processor-behavior analysis, commonly called *low-level analysis*, is the only part of the WCET estimation that is actually dependent on the hardware. It determines local execution times for basic blocks in the program, based on cost models of the architecture under analysis.

An important part of the processor-behavior analysis are the *cache* and *pipeline* analyses. The memory hierarchy has a huge impact on the performance of modern processors. Without a cache analysis, the WCET estimation tool is forced to consider the worst-case scenario, which usually consists in estimating every memory access as a *cache miss*[1]. This often takes one or more orders of magnitude more time than a *cache hit*, leading to a very imprecise overestimation. Pipeline analyses suffer from similar issues.

Several analyses [70] use a data-flow framework based on abstract interpretation to estimate safe approximations of cache and pipeline contents. For instance, a *cache must-analysis* abstracts the contents of the cache with a set of values which *must* be present in the cache at each program point. For such values, a cache hit is ensured, therefore we can safely avoid the penalty for a cache miss without risking underestimating the WCET. The analysis is not exact, so the actual WCET may be smaller, but it is a better approximation than without the cache analysis.

Pipeline analyses operate in a similar way, estimating pipeline occupancy to minimize the number of cache stalls to be included in the worst-case. The downside of processor-behavior analyses is that they are specific to a precise hardware model (processor and memory configuration). In practice, simple microprocessors (such as MCS-51) have accurate timing models (mostly due to lack of cache and pipelines), but obtaining a realistic timing model for more complex architectures constitutes a research and development effort on its own.

3.1.3 Estimate Calculation

The final part of the WCET estimation consists in combining local information obtained by the processor-behavior analysis with global information obtained by the flow analysis. There are different techniques to perform this, namely tree-based, path-based and IPET-based analyses, but

[1] In some architectures, due to a phenomenon called *timing anomaly*, there are situations where combinations of cache hits and misses may perform worse than only having cache misses, which further complicates the analysis.

the latter is by far the most common one, used by the majority of WCET estimation tools. In this section, we describe IPET.

Implicit Path Enumeration Technique (IPET)

The Implicit Path Enumeration Technique [49] uses the execution model of the flow facts we described previously: it associates each program point and edge to a counter representing the number of times the execution along a worst-case execution path reaches this point. IPET then uses flow facts to establish constraints on these execution counters. Each program point has a *cost* which is associated to the results of the processor-behavior analysis. This cost represents the maximum number of cycles for the execution of the program point.

Let x_i and t_i be the execution counters and the cost associated to each program point i, respectively. If we maximize the sum of all such costs in a program, we obtain safe bounds for the WCET:

$$\text{WCET via IPET} = \max \sum_{i \in \text{CFG}} x_i.t_i$$

Figure 3.4 presents an example CFG with timing information displayed to the right of each program point. This information is the cost t_i (in clock cycles) associated to each vertex. Semantic constraints of the form $x_i \leq N$, if any, are represented on the rectangles to the left of some program points. Syntactic constraints are not represented in the figure. In some models ([37, 40]), constraints can be attached to edges ($e_{i,j}$ variables) instead of vertices. Edge constraints can also model the effect of a pipeline analysis [37], where *negative* edge costs indicate that the following instruction is already loaded in the pipeline.

To solve the equation system (which is an integer linear programming problem), we can use any linear programming solver or constraint programming method and obtain the maximum value for the timing estimate function. In the worst case, this has polynomial complexity if there are only flow constraints, or exponential complexity otherwise, but in practice LP solvers handle both systems efficiently. This is one of the reasons the IPET is used by several state-of-the-art WCET tools [70].

An advantage of IPET is that several kinds of flow facts can be specified to improve the precision of the result. For instance, in Figure 3.4, IPET allows constraints such as $x_3 \leq 6$ and $x_5 \leq 2$, which constrain the execution of vertices along several loop iterations. Linear constraints can represent several sorts of flow facts, such as $x_6 + x_9 \leq 10$, which may be the result of a constraint such as *(C_1): program points 6 and 9 are never both taken in the same loop iteration.* The '+' operator acts as an "exclusive-or" concerning distinct program points.

In this last constraint C_1, we see one of the weaknesses of the standard IPET approach: since we represent paths only *implicitly*, and we are using *global* constraints to constrain executions, some precise constraints cannot be expressed, such as *program points 6 and 9 are both taken or both not taken at each loop iteration*[2]. Engblom and Ermedahl [24] detail how to model such constraints, by splitting variables and adding extra constraints on a per-iteration basis. For instance, C_2 could be modeled by replacing x_6 with $x_{6,1}, x_{6,2}, \ldots, x_{6,10}$ (one instance of x_6 per loop iteration) and then adding the constraint $x_6 = \sum_{0 \leq i \leq 10} x_{6,i}$. The same should be done with x_9, and then we might add ten $x_{6,i} = x_{9,i}$ constraints. This solution can be extremely costly in terms of number of variables and inequalities, so in practice this decomposition is either done manually (via user annotations) or using some heuristics which group similar iterations together, to avoid over-generation of constraints.

Finally, yet another benefit of using IPET is that the constraint-based approach enables a separation between constraint generation and constraint resolution. The latter can be performed by external tools, such as off-the-shelf solvers (e.g. LpSolve and CPLEX), and validated for correctness. This will be detailed in Chapter 7.

In Figure 3.4, the final WCET estimated by IPET, considering all semantic constraints, has a cost of 343 cycles (to know whether this result is precise, we would need to look at the program which originated the CFG). The last step of the WCET estimation method concludes with this

[2] A naive translation of this constraint as $x_6 = x_9$ does not work: we may, for instance, take 6 but not 9 in one loop iteration, and do the opposite in the following iteration, which results in $x_6 = x_9 = 1$.

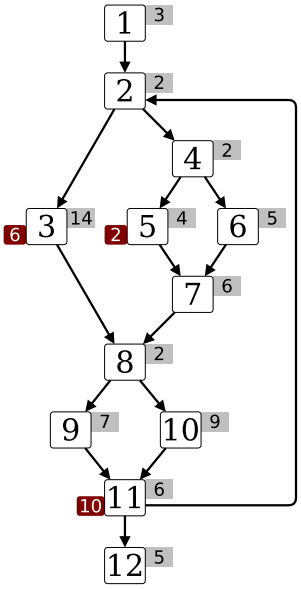


Figure 3.4: CFG with annotated timing costs for each program point (on the right side of each vertex) and some control flow execution constraints (on the left side of each vertex).

numerical value, which is a safe upper bound of the actual WCET. This information can then be used by other components of the development process of the safety-critical system.

3.2 Value Analysis

A *value analysis* is a static program analysis that computes the possible values of program variables. More precisely, it computes, at each program point, and for each program variable, the set of values that may be taken by the variable during program execution, or an over-approximation of this set.

Value analyses have applications in several contexts related to static analysis. For instance, almost all state-of-the-art WCET tools, such as aiT, WCC and SWEET, incorporate some value analysis, using it to obtain ranges for interval variables, information about memory accesses (to perform cache analysis), or to refine control flow information (e.g. identifying dead code). In other domains, value analyses can be useful for compiler optimization (specialized optimization of bounded variables, such as cast elimination), security vulnerabilities analysis (detection of buffer overflows), etc.

In this section, we describe what a value analysis can compute, and how it is performed, using a framework based on *abstract interpretation*. We introduce several terms related to abstract interpretation as applied to a value analysis.

3.2.1 Overview of a Value Analysis

The value analysis we consider here is a particular (but quite common) kind of value analysis, often called *interval analysis* or *range analysis*, where the variable values are abstracted by numeric intervals. Figure 3.5 presents a program along with the result of an interval analysis and the exact values collected by the program semantics. This analysis will be explained in this section. The program P_{VA}, which consists of two loops and some integer manipulations, is based on a program of the WCET benchmark suite used to evaluate our loop bound analysis, however the value k produced by the program is nonsensical. The intervals computed by our analysis are denoted as $[l, u]$, where l is the lower bound (inclusive) and u the upper bound (inclusive). The $\top$ (*top*) symbol indicates an unbounded interval. In practice, this interval is finite because we are dealing with machine integer arithmetic, but it contains all representable integers, e.g. $[-2^{32}, 2^{31} - 1]$ in a signed 32-bit representation. The last column indicates the *actual* values assumed by the program variables,

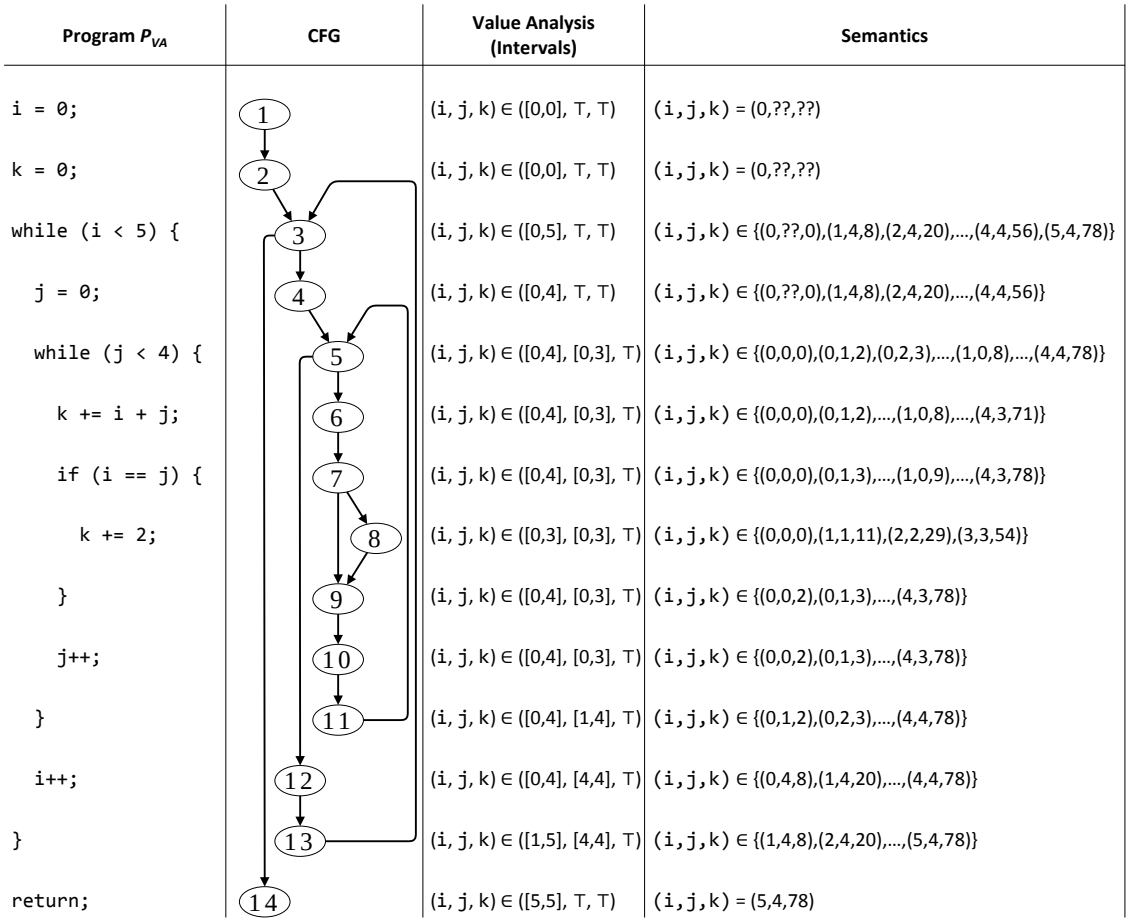


Figure 3.5: Example program P_{VA}, its control flow graph, the result of an interval analysis and the exact semantic sets associated to each program point.

where `??` represents an uninitialized variable.

If we knew the set of all possible execution traces of a program P, we could compute the union of the values assumed by each variable at each program point. This would result in an *exact* analysis. In practice, however, this is unfeasible: it would amount to executing the program for each possible input. And, even if we could do it, we would still need an unreasonable amount of memory to store each execution trace. Using static analysis and abstract interpretation allows us to obtain an over-approximation of the result in a reasonable amount of time, and using a reasonable amount of memory: instead of computing the concrete and possibly infinite result, we compute an abstract information—traces are abstracted via program points, and values are abstracted via intervals—that is finite and tractable. A *correct* analysis returns intervals that are guaranteed to contain *at least* all values collected by the semantics, possibly more. A *precise* analysis returns intervals which contain as few infeasible values as possible.

In Figure 3.5, the third column presents the result of the interval analysis *before* each program point, that is, they correspond to the *union of values from incoming edges*. For instance, at program point 3, variable i is associated to the interval $[0, 5]$, which is the union of the values coming from its two predecessors, 2 ($[0, 0]$) and 13 ($[1, 5]$). Finally, the last column shows the exact values collected by the program semantics, as triples (i, j, k) of values, or as sets of such triples, for program points inside loops.

From Figure 3.5, we can (informally) check the correctness of the interval analysis by verifying if, for each value present in the last column, this value is within the bounds of the corresponding interval in the value analysis. For instance, if we project the first element of the triples collected by the semantics (which corresponds to variable i) at each program point, obtaining a set of values

per program point, we can see that this set is entirely contained in the interval given by the value analysis. This is what we call the *soundness criterion* of an analysis, that is, its ability to predict results which are correct with respect to the formal semantics of the program. We can also see that the analysis is *precise* concerning variable i: its intervals are *tight*, that is, they contain no values which are not effectively present in the semantics.

For variables j and especially k, however, the value analysis is less precise. For k in particular, the analysis does not infer any useful interval. A trivial value analysis which infers $\top$ for every program variable is indeed correct, but not very useful in practice. Therefore, besides proving the correctness of the analysis, it is also important to either prove its precision or, more commonly, to test its result on some benchmarks, to ensure the value analysis actually provides useful output. For instance, the design of the Astrée [18] static analyzer, which includes a value analysis, has been performed using a refinement strategy, starting from a simple and fast (but imprecise) version that is then incrementally improved with the addition of more and more complex abstractions, until it reaches an acceptable cost/precision ratio.

3.2.2 Value Analysis via Abstract Interpretation

The standard way to define a sound value analysis is by using the framework provided by *abstract interpretation* [17], which is a general theory of sound approximations of program semantics. Abstract interpretation is a widely used technique in the formalization of static analyses due to its solid mathematical bases. In this manuscript, we do not detail all the abstract interpretation-related aspects underlying the soundness of the value analysis; instead, we describe it from the point of view of a user of such a framework. In this section, we introduce several concepts related to abstract interpretation, which will be used in Chapter 6, where we present and evaluate a formally verified value analysis.

Orders and Lattices Let S be a set. In our analysis, we will consider sets of machine integer (integers modulo 2^{32}) intervals, where an interval is a pair $[l, u] \in \texttt{Int} \times \texttt{Int}$, with l being the lower bound and u its upper bound (which implies $l \leq u$). We also consider the empty interval, noted $\bot$, which is a normalized version of all intervals $[l, u]$ where $l > u$. `Int` denotes the set of 32-bit long machine integers. These integers can be represented either as *signed* values, those in the set $\{z \in \mathbb{Z} | -2^{31} \leq z < 2^{31}\}$, or as *unsigned* values, in $\{z \in \mathbb{Z} | 0 \leq z < 2^{32}\}$. We will usually consider their signed representation, unless explicitly stated otherwise.

A *partial order* $\sqsubseteq$: $S \times S$ is a binary relation on S that is reflexive, transitive and antisymmetric. Our partial order will be the interval inclusion, noted $\subseteq$: an interval $a = [l_a, u_a]$ is considered "smaller than" another interval $b = [l_b, u_b]$ iff a is included in b, that is, $l_b \leq l_a$ and $u_a \leq u_b$. Note that this order is *partial* since some elements cannot be compared, such as $[1, 3]$ and $[0, 2]$. Note that the $\subseteq$ relation represents the *precision* of an analysis: a smaller solution contains fewer spurious elements.

We will use two additional binary operators defined on our sets S:

- a *join* operator (also called *least upper bound*, abbreviated as *lub*), noted $a \sqcup b$. This operator returns the smallest (according to the partial order $\sqsubseteq$) element in S that is greater than both a and b;
- a *meet* operator (also called *greatest lower bound*, abbreviated as *glb*), noted $a \sqcap b$. It returns the largest element in S that is lower than both a and b.

These operators are associative, commutative and idempotent. In our analysis, the join operator corresponds to the *convex union of intervals*, defined as follows: $[l_a, u_a] \cup [l_b, u_b] = [\min(l_a, l_b), \max(u_a, u_b)]$, while the meet operator corresponds to the *intersection of intervals*, defined as: $[l_a, u_a] \cap [l_b, u_b] = [\max(l_a, l_b), \min(u_a, u_b)]$.

A partially ordered set (*poset*) is a set S equipped with a partial order: $(S, \sqsubseteq)$. A *lattice* is a poset where each pair of elements has a least upper bound and a greatest lower bound. A *complete lattice* has a greatest element, noted $\top$ (*top*), and a least element, noted $\bot$ (*bottom*).

In our analysis, the bottom element $\bot = \emptyset$ is the empty interval, and the top element $\top$ is the interval containing every possible machine integer. For convenience, it is represented as $[-\infty, +\infty]$, though a more accurate representation would be $[-2^{31}, 2^{31} - 1]$ for signed intervals.

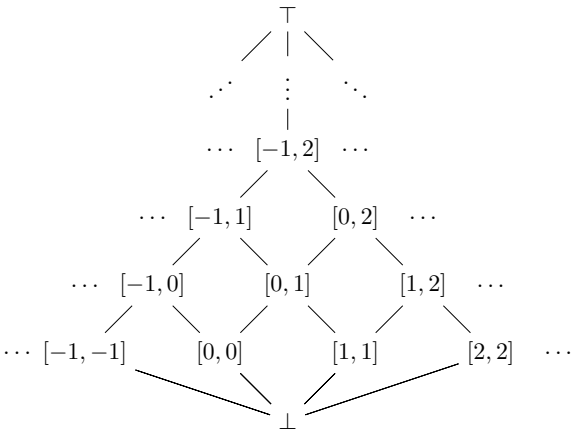


Figure 3.6: Hasse diagram of the lattice of integer intervals used by the value analysis.

Figure 3.6 depicts a simplified version of the lattice of integer intervals used by our value analysis. This diagram, called *Hasse diagram*, organizes the lattice elements according to the partial order: if an element a is lower than an element b, then a appears below b in the diagram, and there exists an ascending path from a to b. Just above the bottom element, we have singletons; above them, intervals containing 2 adjacent values, and so on.

Transfer Functions, Fixpoints, Widening and Narrowing In data-flow analysis, a *transfer function* is a *monotone*[3] function $f_t : L \to L$, where L is a complete lattice. A *monotone function* is a function that preserves order, that is, for f_t monotone, if $x \sqsubseteq y$, then $f_t(x) \sqsubseteq f_t(y)$. Monotone functions, when applied to complete lattices, imply that the equation system always admits one least solution. Another consequence of monotonicity is that more precise inputs to the analysis are ensured to produce more precise outputs.

In our value analysis, the transfer function corresponds to the evaluation of a program statement or expression, and it maps the abstract state before evaluation to the abstract state after the evaluation. When dealing with elements that might repeat an unbounded number of times (in loops and recursive function calls), we need the concept of fixpoints. A *fixpoint* (also called *fixed point*) of a function $f : L \to L$ is a point $x \in L$ such that $f(x) = x$.

We can graphically represent the lattice of abstract semantic states, as in Figure 3.7, to better understand how the analysis progresses. The lattice is represented by the outer oval, with two subsets of special interest highlighted in the figure: the area of *post-fixpoints* (upper gray oval) and the area of *pre-fixpoints* (bottom white area). The latter contains elements which increase (with respect to the lattice ordering) when f is applied to them, while the former contains elements which decrease. The central area contains fixpoints, stable under application of f. The "ideal" result is the *least fixed point* (lfp), the bottom of the central gray lattice, because this is the most precise (smallest) result.

Elements in the white zone do not constitute correct solutions of the analysis. For instance, $\bot$ is not a solution of any meaningful program. On the other side, elements in the gray zone are correct, albeit sometimes imprecise, solutions. Those in the central gray area are correct and stable (through application of f) solutions. The best (most precise) solution is the one represented by the *lfp*, but sometimes, when the lattice is huge, it cannot be computed in a reasonable amount of time.

Initially, we have no information about any program point, so the analysis starts at the bottom of the lattice. Following Figure 3.7, we apply f once, but do not obtain a fixed point. We then apply f again $n-1$ times (n might be quite large), but do not obtain a fixed point. Convergence is not guaranteed, especially if the program contains loops or recursive calls. In this case, an operator called *widening* (noted $\triangledown$) can be used to speed up convergence of the iteration. This operator has the property that a finite sequence of applications of this operator is guaranteed to stabilize on a value that is a safe approximation of the least fixed point, that is, somewhere in the gray zone.

[3]Transfer functions are not *required* to be monotone, but in practice they almost always are.

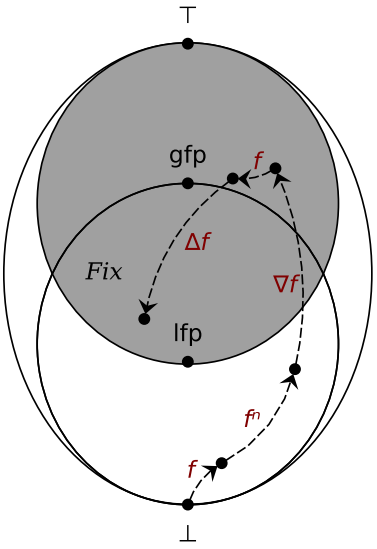


Figure 3.7: Lattice of a domain L and the evolution of a function $f : L \to L$ that represents our static analysis. It evolves through n successive applications of f, then a widening, another application of f and finally a narrowing. The final element, a fixpoint, is the result of the analysis.

This guarantees correctness at the price of precision: in the worst case, the widening might end up with $\top$ as result.

Finally, another operator, *narrowing* (noted $\triangle$) can be used to improve the precision of the previous result: the narrowing operator results in a (safe) state which is not greater than the original one. Widening and narrowing operators are useful to speed up and improve the precision of the value analysis. The widening operator can be seen as some sort of heuristic, and since it only produces over-approximations, it usually has little cost in terms of proof effort for correctness. Indiscriminate use can however lead to bad overapproximations. In practice, both widening and narrowing are alternated to balance efficiency and precision of the analysis.

3.3 Related Work on WCET Estimation and Compilation

We present here a state-of-the-art survey on static WCET estimation tools, WCET-related compilers, and similar approaches related to the work performed in this thesis. This is not an exhaustive overview; also, some of the mentioned tools, such as CerCo (Section 3.3.5), present similarities but are directed towards fundamentally different objectives.

3.3.1 SWEET

SWEET [26] is a research prototype for flow analysis and WCET estimation developed mainly by researchers at the Mälardalen University, Sweden. It uses abstract interpretation-based static analysis for its timing estimations. Its automatic loop bound estimation method has served as inspiration for our own loop bound analysis (described in Chapter 4).

The input to SWEET are files in the ALF (ARTIST2[4] Language for WCET Flow Analysis) format, a low-level intermediate language intended to be reused among different tools as a common format between compilers and timing analysis tools. There are tools which perform the compilation from C code to ALF, developed to be used in conjunction with SWEET. These tools produce a mapping from the C source to the intermediate representation, which is the format used in the annotation files specifying flow facts.

Flow analysis is one of the main strengths of SWEET: besides having a very powerful flow fact annotation language, in part due to the possibility of manually annotating the ALF code, it has

[4]ARTIST2 is a European Network of Excellence research program.

an automatic loop bound analysis which can infer bounds for different kinds of loops (including triangular loop nestings). It is based on a combination of abstract interpretation, program slicing and value analysis.

Like most WCET estimation tools, SWEET's architecture follows the three-stage method described in Section 3.1. The first stage is the flow analysis, which produces flow facts as annotations. Then, the annotated ALF file is given to `low_sweet`, which performs the processor-behavior analysis (including cache and pipeline analyses) and returns a timing cost for each basic block in the program. The actual IPET estimation is computed either via IPET, or via SWEET's *abstract execution* [26] mechanism, which acts as an interpreter, executing each basic block and adding its estimated execution time to a ghost variable containing the global estimated WCET. Optionally, instead of using `low_sweet`, the flow facts can be fed to other tools, for instance aiT and RapiTime[5], where the generated flow annotations replace those manually input by the user of these tools. In this context, SWEET only performs the flow analysis stage, delegating the rest of the WCET estimation to the other tool.

Relation to the work in this thesis Our loop bound estimation technique is inspired from SWEET's method, but it is formally verified. The extra effort needed due to the formal verification process results in differences of precision between the tools, since our analysis does not incorporate all of the most advanced aspects of the value analysis, namely a pointer analysis and congruence information (also called *strided intervals*, used to represent sets such as $\{0, 4, 8, \dots\}$). Another difference between the methods is that SWEET analyzes each loop iteration separately, which is more precise but more costly in terms of analysis time (the analysis of some programs does not terminate in practice due to the increased computation time).

Mälardalen WCET Benchmarks The Mälardalen WCET benchmarks [34], gathered at the Mälardalen Real-Time Research Center (the same authors of SWEET), consist mainly in a series of stand-alone C programs, plus loop bound information related to these programs in a separate file. This loop bound information can either be used as input for low-level WCET analyses (whose objective is to estimate the WCET, supposing the flow analysis has been performed by someone else, either a person or a tool), or it can be used as oracle (expected output) for flow analyses. This set of programs is the reference benchmark in the WCET community and often cited to compare the precision of different WCET estimation techniques.

The selected programs, collected from a series of research groups and industrial tool vendors, are mostly oriented towards embedded systems programming: there is no dynamic allocation and no recursive calls (except for one program). To allow for precise binary-level comparisons and to simplify installation, the programs include no external libraries (some functions are abstracted by stubs).

The programs in the benchmark suite range from a few dozen to a few thousand lines of code. They include various types of control flow structures: simple `for` loops with static bounds, `while` loops depending on memory accesses, complex conditions with array elements, triangular loop nests, etc. Some benchmarks are single-path programs, others have input-dependent control flow. Provided input annotation files define intervals for the initial values of some variables, forcing the analysis to consider multiple paths.

These benchmarks were used by the developers of SWEET to evaluate the precision (and also the performance, via the time and memory consumption) of SWEET. Other authors used them to compare their flow analyses to SWEET's. In this thesis, we evaluated our loop bound analysis and compared it to SWEET's using these benchmarks (this evaluation is described in Section 7.3.1).

3.3.2 aiT

The AbsInt Timing analyzer [29] is a timing analysis tool developed and commercialized by AbsInt Angewandte Informatik, a German company. It is part of the a^3 (AbsInt Advanced Analyzer) framework. aiT is based on abstract interpretation, hence its name, and it is the reference commercial

[5]RapiTime is an estimation-based timing tool commercialized by Rapita Systems Ltd. RapiTime is not detailed in this survey because it is not a *static* timing tool, therefore its results are not guaranteed to be sound.

timing analysis tool in the domain of embedded systems, having been interfaced with several other tools (SWEET and WCC, to cite a few).

The input to aiT is a binary (executable) program, plus possibly an *annotation file*, containing manual annotations which complement the static analysis performed by the tool. These annotations can indicate loop bounds, flow facts, targets of dynamic branches (e.g. jump tables), register and variables values, etc. They are useful both for control flow and processor-behavior analysis.

The first step of aiT's analysis is the reconstruction of the control flow graph. Then, a value analysis is performed for the classification of memory accesses, which is useful for the cache and pipeline analyses. The analysis then proceeds as indicated in the general WCET estimation framework: flow analysis, processor-behavior analysis (including a cache analyzer, whose results are fed into a pipeline analyzer), combination of results for the IPET-based path analysis, and final computation of the WCET estimation. Some of the main strengths of aiT are that, being an industrial tool, it has an accurate timing model of complex architectures (including realistic processors currently used in the industry, such as PowerPC), it has a wide variety of hardware targets, and it is actively maintained.

The flow analysis performed by aiT includes a loop bound estimation. This stage relies on the results of the value analysis and on pattern matching [29] (e.g. recognizing common loop patterns, such as `for` loops of the form `for (i = 0; i < N; i++) {...}`). This loop bound analysis is claimed to only work for simple loops, and its soundness may depend on user feedback: for some kinds of loops, the analyzer generates alerts about assumptions it needs in order to ensure the produced loop bounds are correct. It is up to the user to verify these assumptions are valid.

aiT has a very rich annotation language, including different kinds of behaviors for user annotations: some of them behave as *verification conditions*, meaning that the user wants the analysis to report an error if it cannot check them, while others behave as *trusted assertions*, meaning that the user is supplying information which should be considered true by the analysis. The former is used to provide feedback to the user but does not impact the correctness of the analysis, while the latter adds the user assertions to the trusted computing base. This means the analysis is unsound if the user provided invalid assertions.

aiT's WCET estimation supports recursive calls, on the condition that maximum recursion depth is supplied by manual annotations. aiT's automatic loop bound analysis does not handle irreducible loops. They can only be handled via manual flow annotations.

Being a commercial tool, not many details of aiT's abstract domains are disclosed. Its value analysis is claimed to be interval-based [29], which seems to indicate no relational domains are used. There is no mention of any special annotations or particular handling applied to nested loops, e.g. triangular loop nestings.

Relation to the work in this thesis aiT is the leading example of an industrial-strength timing analysis tool. It illustrates the fact that very complex analyses (using relational abstract models, termination analysis, etc.) are not necessarily the most essential component for an industrial usage. In this context, flexibility to accept several kinds of user annotations and detailed hardware models are imperative. Unfortunately, this requires the use of untrusted annotations.

aiT can be seen both as a back-end to the automatic flow analysis performed in this thesis (in which case it would have to be added to the trusted computing base) or as a guide to which analyses should be formalized by a verified timing tool: cache analysis, pipeline analysis, etc.

3.3.3 Heptane

Heptane [15] is a research prototype for static WCET estimation. It focuses more on hardware features than on flow analysis. It includes modern architectural elements, such as shared caches for multi-cores and multi-level caches. It also includes a pipeline analysis and supports several architectures (primarily MIPS and PowerPC, but also ARM). Heptane can be used in combination with cross-GCC, which enables it to take as input a C source program, which is then cross-compiled into the target architecture and analyzed. Alternatively, it takes as input the assembly code.

Heptane's flow analysis stage includes a CFG reconstruction from the binary file, but no automatic loop bound estimation, therefore loop annotations must be input manually. The user may insert them directly at the source level, in which case they are passed to the assembly code.

Loop bound annotations are accepted anywhere inside a loop. The bound estimation phase is based on IPET. Heptane enables the configuration of several elements of the target architecture via a configuration file. For instance, the definition of the cache hierarchy (both data and instruction caches), number of lines, latency, pipeline computation method, etc. This facilitates comparisons between architectural variants and it also enables the deactivation of some analyses.

Relation to the work in this thesis Heptane's WCET analysis is complementary to the one performed by our tool: it specializes in processor-behavior analysis, while ours is focused on the flow analysis. Combining both tools is certainly useful to obtain more information about WCET estimations. To confirm the feasibility of an integration between our tool and Heptane, I combined the result of the loop bound estimation provided by our tool with Heptane's WCET estimation. Although the WCET estimate in this case is not formally verified, it will be used in the future (e.g. after a cache analysis is implemented in our tool) to compare the estimates provided by our tool and those provided by Heptane. Since Heptane is not formally verified, this integration improves confidence in its results and minimizes manual intervention, thanks to the automatic loop bound estimation which replaces manual annotations.

3.3.4 WCC

The *WCET-aware C Compiler* [28] is a C compiler whose main purpose is to perform WCET-oriented optimizations[6] to lower the final WCET of the produced code. It is tightly integrated with aiT, which acts as a back-end producing the actual WCET driving the optimizations.

This compiler works using an estimated WCET as value to be optimized by the compilation. By varying the parameters and heuristics on some predefined optimizations, such as procedure cloning and memory positioning of program code and data, the compiler produces code with different estimations for the worst-case time. It obtains these estimations by sending the compiled code to aiT and asking it to perform the WCET estimation. It then modifies some parameters (e.g. number of loops to unroll, procedures to clone, etc.) using machine learning techniques (i.e. using the WCET estimation as the objective function to be minimized), compiles a different code, and sends it to aiT again. The final output of the compiler is the code with the smallest WCET estimate.

With respect to its flow analysis, WCC integrates flow fact annotations (including loop bounds and more general annotations, similar but less powerful than those provided by SWEET) and an automated loop bound analysis [50], based on abstract interpretation and program slicing, like the one provided by SWEET. The main differences are that, here, the abstract interpretation is applied at the source level (instead of an intermediate language like SWEET's ALF format), and that WCC uses a *polyhedral loop evaluation* to compute loop bounds, for loops meeting a set of syntactic criteria (e.g. condition statements must be affine expressions). This analysis consists in using polyhedra to represent the evolution of loop iteration variables and then counting the number of points inside the polyhedra. For instance, we could apply it to the loop in Figure 3.3, where the polyhedron is a triangle containing 15 points, which means it can be bounded to 15 loop iterations. The syntactic restrictions imposed on the shape of the loop body ensure termination: all assignments to the loop iteration variable must be of the form "increment (or decrement) by a known value". If this results in the iteration variable not changing at all, then loop evaluation will fail. The authors claim a very high success rate with their loop bound analysis (99% of the loops in their benchmarks are bounded), using an interval domain.

WCC performs a mapping of the source code to the binary code, enabling it to transform flow fact annotations obtained at the source level into flow facts at the binary level. These facts are then used by aiT to estimate the WCET, without need for manual user intervention. Recursion is supported by WCC, but in general requires manual flow fact annotations specifying recursion depth.

Despite being a compiler with integrated analysis for WCET-related applications, WCC's focus is not on mechanical correctness, but on optimizing the program with WCET-based metrics. One of its interesting features is its *back-annotation* mechanism, which can transpose flow fact annotations

[6] Most optimizations consist in standard compiler optimizations, such as loop unrolling, extended with an estimation and comparison of the WCET of the original and transformed programs. The optimization is applied only if the new WCET estimation is better.

to the source code. This allows, for instance, to automatically obtain as many bounds as possible, and then annotate the source code with them, to allow the user to fill in the bounds which could not have been automatically determined.

Overall, the fact that WCC's loop bound analysis applies techniques similar to those used by SWEET's (program slicing and abstract interpretation) indicate that this approach is adaptable to different contexts. However, the lack of focus on the flow analysis makes it more difficult to obtain enough in-depth information about WCC's method, to mechanically verify its correctness.

Relation to the work in this thesis WCC's loop bound analysis, integrated in a compiler, and using a framework similar to the one used by SWEET, confirms that such a development is relevant and it can be extended with the purpose of not only ensuring the correctness of the timing estimates, but also to optimize them. This is a very distant target, however, since it would include all the aspects related to the hardware modeling and verification of the WCET itself. Still, it confirms the tendency to combine compilation and timing tools as a means to obtain more information about the program, resulting in a more precise estimation.

3.3.5 CerCo

Certified Complexity [5, 4] is a research project, developed from 2010 to 2013, whose main contribution is the back-annotation of *cost* information, from a compiled program back to the source code. The WCET estimation is such an example of a cost annotation. The back-annotation process has been formally verified using the Matita proof assistant: from a compiled code and a cost model (such as the hardware timing model mentioned in Section 3.1.2, used for the processor-behavior analysis stage of WCET estimation), the generation of annotations on the assembly code is transported back to the C level.

More concretely, CerCo starts with a C program and compiles it to assembly code, injecting annotation points in the program. These points are then filled with cost information obtained from the hardware timing model. Currently, the modeled architecture in the CerCo project is an 8051 Intel 8-bit micro-controller, with constant cost instructions, without cache or pipeline, which allows for a simple *compositional* cost model: each function and code fragment can be analyzed independently, and the final cost is the interval sum of the parts. For instance, if a code contains a branch, then two costs will be produced, a minimal and a maximal one, resulting in a *cost range* instead of a single cost. This information is then mapped back into the source. Afterwards, using non-formally verified components, the C source annotations are given to a tool based on program proof, Frama-C[7], which uses them to reason about the cost information. For instance, a parametric WCET estimation based on input variables can be given for some code fragments such as functions. An advantage is that such bounds have *precision* guarantees with respect to the exact value (up to a constant factor); however, they are not formally verified, relying on Frama-C and its deductive verification for correctness.

This process is not entirely automatic: especially in the presence of loops, the generated verification conditions cannot always be automatically proved. In this case, the user must manually complete the proof steps, for instance by giving an explicit loop invariant.

Relation to the work in this thesis The work performed in CerCo shares some similarities to the work in this thesis (integration of compilation and worst-case estimation; use of formally verified compilation; incorporation of external hardware timing models), however the objectives and means are quite different: in this thesis, we present some techniques and ideas concerning the formal verification of existing WCET-related methods, which closely resemble the approaches used by state-of-the-art tools. The CerCo project proposes a different approach, where information is obtained at the hardware level but then transposed to the C source.

From an abstract point of view, both CerCo and the work in this thesis, and also several other works in the literature have the same long-term goal: obtaining precise and provably correct bounds for worst-case execution time. But what distinguishes them are the more concrete goals, as well

[7] Frama-C is a platform for static analysis of C code. It includes several analyses implemented as *plug-ins*, such as a deductive verification mechanism using automated theorem provers.

as the employed techniques: use of a proof assistant plus Hoare logic-style assertions (including manual annotations to aid the proof of some complex invariants) for CerCo, versus a centralized specification and proof of correctness using a single proof assistant (Coq) in our case.

3.3.6 TuBound and r-TuBound

TuBound [59] is a WCET estimation framework composed of several independent tools. Its main objective is to allow source-level annotations to be passed on to assembly-level code even in the presence of non-trivial optimizations, that is, transformations of the source code which might affect the semantics of the annotations (e.g. loop unrolling can invalidate loop bound annotations). TuBound includes an automatic loop bound analysis based on an interval analysis followed by a constraint-based approach. This is similar to a pattern-matching approach, in which suitable loops are checked for syntactic features (e.g. there must be a single loop exit condition; the increment to the iteration variable i must have the form $i := i + c$, for a constant value c; etc.), except that each requirement is defined as a constraint and given to a solver. Using the solver abstracts away and simplifies some implementation aspects. In particular, it results in better bounds for nested loops, because the constraint solver can apply a *labeling*[8] to count the sizes of variable domains, instead of simply multiplying the domain sizes. Triangular loops can thus be bounded exactly with this approach. The downside is that the solver becomes part of the trusted computing base.

Formally proving a loop bound analysis based on constraint programming requires justifying the generation of each constraint. Since most of them are related to the syntactic pattern-matching, this approach requires some complex invariants to relate these properties to the language semantics. For instance, one of the constraints states that *in the loop body, excluding the exit condition and the loop increment, the iteration variable i does not appear on the left-hand side.* It is then necessary to show that this constraint, combined with all others related to the iteration variable, imply that the loop terminates and that the number of iterations can be given by a simple formula.

TuBound's loop analysis has been extended in r-TuBound [39], which uses recurrence equations[9] to establish relations between loop variables. Solving these equations allows us to obtain bounds for loops, by finding the smallest n (iteration number) for which $i(n+1) > b$. Non-linear increments can be dealt with by using solvers (e.g. SMTs) which reason over non-linear arithmetic.

r-TuBound also introduces *control flow refinements*, that is, special cases where loops with complex control flow (such as loops with multiple exits, or multiple increments of the iteration variable) can be bounded. This allows more loops to be bounded, but unfortunately this approach becomes similar to pattern-matching, where each extra loop shape requires a specific reasoning and algorithm, which do not accommodate much variation. This is justified by the fact that most loops in WCET benchmarks conform to few loop shapes, therefore such effort might be worthwhile.

Besides the automatic loop bound estimation, TuBound incorporates a source-to-source transformer which performs loop optimizations and transforms the loop bound annotations consistently. Afterwards, an annotated C program is produced, and compiled with a modified GNU C compiler which preserves the control flow and the bound annotations. The code is compiled into assembly for a specific microprocessor, the Infineon C167. Finally, a WCET estimation tool, CalcWCET$_{167}$, operates on the assembly code to produce the WCET estimation.

Relation to the work in this thesis TuBound is a WCET estimation framework which combines an automatic loop bound estimation and compilation to obtain WCET bounds. It is not verified and, due to the usage of several different tools which have not been originally conceived as a whole, it may contain bugs in the interface between the components, which are typically hard to detect. For instance, the interval analysis is performed by a tool, and then reused by the loop bound estimation, which does not operate on the exact same source code (due to normalization).

TuBound reinforces the idea that compilation and WCET estimation are closely related and integrated tools are necessary. However, from the point of view of formal development, its techniques

[8] In constraint logic programming, a *labeling* of a given set of variables is an enumeration of the sets of values of such variables which satisfy all the constraints. This corresponds to enumerating the domains of iteration variables. For instance, for constraints $0 \leq x < 3$ and $2 \leq y < 4$, labeling produces $[(0,2),(0,3),(1,2),(1,3),(2,2),(2,3)]$.

[9] An example of a recurrence equation, for a loop such as `for (i=a; i<b; i=c*i+d)`, would be $i(n+1) = c * i(n) + d$, with initial value $i(0) = a$.

seem hard to verify or validate *a posteriori*.

3.3.7 OTAWA and oRange

OTAWA [6] is a WCET estimation framework designed from the start to support different WCET-related tools, instruction sets and processor architectures, allowing comparisons between them. It takes a binary program as input and provides a uniform program representation for its several tools. Among them, there is oRange [52], a tool for loop bound estimation of C sources that will be integrated into OTAWA. So far, it produces annotations which are transmitted to the compiled code via debugging information. The compiler itself is not integrated into OTAWA.

oRange incorporates abstract interpretation, loop normalization and recurrence relations to bound loops. A first step consists in identifying loop iteration variables and loop increment statements. This is done by using abstract interpretation to compute a map from variables to expressions. This map is used to identify the iteration variable[10] among all variables assigned in the loop. For instance, in a loop such as `for (i=3;i<=n;i+=2);`, variable i is identified as the iteration variable, with initial value 3, step 2 and upper bound n. The number of iterations is therefore $\lfloor \frac{n-3}{2} + 1 \rfloor$ for this loop. Note that the loop bound expression is parameterized by n. This is important for nested loops, where expressions may depend on several variables. The following step consists in normalizing all loops, simplifying the associated expressions, and then evaluating these expressions for two variables per loop: a *max* counter which establishes the maximum bounds *per loop entry* (in other words, a local bound) and a *total* counter which accumulates the iterations from all passages through the loop (a global counter).

There are some similarities between the loop bound estimation of OTAWA and r-TuBound, in that both are based on recurrence relations and compute precise bounds for triangular loops based on the values of the expressions related to the iteration variables. For instance, both mention that Computer Algebra Systems (e.g. Mathematica and Maple) might be used to compute the expressions.

Besides loop bound estimation, OTAWA includes abstraction layers for the instruction set and for the hardware architecture. The former enables most components to be designed independently of the instruction set (e.g. PowerPC, ARM, Sparc, etc.), while the latter allows different hardware configurations (e.g. number of cache lines, memory latency, pipeline stages, etc.). The WCET estimation is obtained by sequentially composing the analyses: CFG reconstruction, loop analysis, cache analysis and IPET. OTAWA also includes a SystemC-based architecture simulator (based on a hardware description file), used to evaluate the precision of the estimated WCET.

Relation to the work in this thesis OTAWA does not include a compiler and the integration of oRange so far relies on debugging annotations. The authors mention a possible integration with GCC, which could bring improvements to the loop analysis. oRange's loop bound estimation is automatic but not formally verified. Verifying it at the source level would be an ambitious endeavor, due to the size and complexity of the C language.

The *max* and *total* constraints are closely related to the local and global counters used in the instrumented Cfg semantics (presented in Section 4.4.1) of our loop bound formalization. The main difference between the two techniques is that oRange uses abstract expressions based on iteration variables, which are more precise (especially for complex and nested loops) but more complex to reason about. They are also more restrictive (e.g. each loop must have a single iteration variable), although the loop normalization step eliminates some constraints. In our method, one might argue that program slicing plays a similar role. From the point of view of formal verification, OTAWA's strength (the relative independence of its several tools) is a hindrance to having it formally proved, due to the problems that arise in the interfaces between components.

3.4 Conclusion

The existence of a common architecture for several static WCET estimation tools allows us to present the overall workings of our method in a larger context. We do not propose a radically

[10] This method only works for loops containing a single iteration variable.

innovative solution, but instead we perform the consolidation, through formal verification, of a technique known to work well. In particular, we focus on the flow analysis stage, proposing an automatic loop bound estimation method to avoid manual annotations and to minimize the trusted code base. One of the components of the loop bound estimation is a value analysis based on abstract interpretation, which is a domain known for relying on several concepts and definitions which require some explanations. The pragmatic primer on abstract interpretation presented in this chapter aims to explain some aspects of the value analysis mentioned in Chapter 4. More importantly, this introduction to value analysis is important for the detailed value analysis in Chapter 6.

Through the survey on WCET estimation, we observed that formal verification is still quite distant from the reality of most WCET estimation tools. On the other hand, manual loop annotations is increasingly replaced with automatic estimations. Several tools provide such estimation of loop bounds, with different methods and varying results. Some of these methods, such as the one proposed by SWEET, are more adapted for a formal verification effort than others. In particular, we chose this method as basis for our verified development effort, which is the subject of the next three chapters in this thesis.

Chapter 4

Loop Bound Estimation

The main component of our WCET estimation tool is an automatic loop bound analysis. This analysis is based on the method of the SWEET [35] tool. Our contribution is to have formally verified each component of this analysis. In this chapter, we first present the method in Section 4.1. It is composed of three main parts: program slicing, value analysis and loop bound estimation. Program slicing is formalized in Chapter 5, and the value analysis is formalized in Chapter 6. We focus here on the loop bound estimation, including a formalization of the notions of local and global bounds (Section 4.2), the pseudocode of the estimation method (Section 4.3) and its correctness proof (Section 4.4). The proof is based on the semantics of the ICfg language, instrumented with execution counters. The formal Coq development of the proof is available online[1], and its presentation here is for ease of understanding. An experimental evaluation of this technique, including a comparison with SWEET, is presented in Section 7.3.

4.1 Presentation of a Loop Bound Analysis for WCET

The loop bound estimation method we formalized in this thesis is based on the estimation performed by the SWEdish Execution Time tool (SWEET [35]), which was described in Section 3.3. This analysis automatically computes safe loop bounds for C programs, a necessary step for the WCET estimation as we have seen in Section 3.1.3. In this section, we present the method without formalizing it. The following sections will present the formalization.

4.1.1 Basis of the Method: Pigeonhole Principle

The loop bound estimation is based on the *pigeonhole principle*, which fundamentally enables counting the number of loop iterations. This principle, in its general form, states that:

> If n pigeons are put in m pigeonholes, where $n > m$, then at least one pigeonhole must contain more than one pigeon.

In the loop bound estimation, a "pigeon" corresponds to a reduced program state, that is, a set of values for (some) program variables. A "pigeonhole" corresponds to a counter of the number of occurrences of a given element in the execution trace, that is, if the same element appears twice, then both occurrences are put in the same pigeonhole.

In other words, SWEET's loop bound estimation is an application of the pigeonhole principle to the execution trace of a terminating program, whose assumptions are based on the following two facts:

- in a deterministic and terminating program, the same *variable state* never arrives twice;
- therefore, a safe bound for the number of executions of a given program point is the number of possible different states at that point.

[1] The Coq development of the loop bound estimation is available at http://www.irisa.fr/celtique/ext/loopbound.

The first fact explains *why* the method works, and it is the consequence of a deterministic environment: should the same variable state happen twice in the execution trace, this would mean that the program does not terminate: if a program steps through states σ_1 then σ_2, then its determinism implies that σ_2 is the only possible successor to σ_1. Therefore, if σ_1 is ever reached again, then σ_2 will be its successor, and so on, indefinitely. Such a program forcibly diverges. Since we had supposed a terminating program, this entails a contradiction, and therefore such a situation (repeating occurrence of a program state) cannot happen.

The second fact gives information related to *how* the method works: the number of states is bounded, and therefore the number of different values for program variables is also bounded. The pigeonhole principle converts bounds related to program variables into bounds related to execution counters. Further refinements (such as the use of program slicing to reduce the number of variable values to be considered) improve the precision of the method, but its correctness only depends on these two fundamental assumptions.

4.1.2 Application on an Example

The program in Figure 4.1 is adapted from a LU decomposition algorithm, simplified to make it smaller. It has been chosen because this is one of the simplest benchmarks containing nested loops that also has other interesting features relevant to the loop bound analysis. The program consists of some variable initialization, then a loop that iterates over all elements of a, in row-major order, storing the result of some calculation on each element. At the same time, variable w accumulates the sum of the elements in each row and stores it in vector b.

The control flow in this program is composed of two nested loops and a nested `if`. There are several data dependencies related to the iteration variables, especially i, but no dependencies related to b. There are memory loads and stores, but no function calls.

Program *P*	CFG with Loops	Slice P_1' w.r.t. 1st loop (at 4)	Val. Analysis for P_1' (VA_{IN})	Domain Sizes	Π(Sizes)
`n = 5;`	1	`n = 5;`			
`i = 0;`	2	`i = 0;`	n:{5}		
`w = 0.0;`	3				
`do {`	4	`do {`	n:{5}, i:[0,5]	n:1, i: 6	1 × 6 = 6
`  j = 0;`	5				
`  do {`	6				
`    a[i][j] = i+1+j+1;`	7				
`    if (i==j)`	8				
`      a[i][j] *= 5.0;`	9				
`    w += a[i][j];`	10				
`    j++;`	11				
`  } while (j≤n);`	12				
`  b[i] = w;`	13				
`  i++;`	14	`  i++;`	n:{5}, i:[0,5]		
`} while (i≤n);`	15	`} while (i≤n);`	n:{5}, i:[1,6]		
`return;`	16	`return;`	n:{5}, i:{6}		

Figure 4.1: Simplified LU decomposition program used as example for the analysis. From the third column on, we have the application of the method on the first (outer) loop: program slicing, value analysis and bound calculation.

4.1.3 Method Decomposition

The loop bound analysis can be decomposed in these steps, illustrated in Figure 4.2:

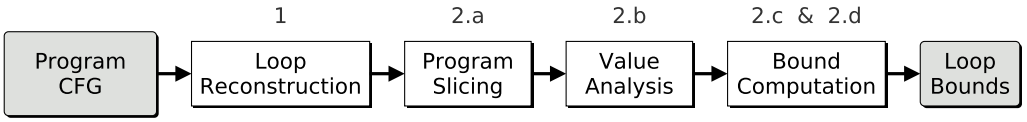


Figure 4.2: Architecture of the loop bound analysis.

1. Loop extraction: reconstruction of the loop structure of the program from the CFG;
2. Individual loop analysis, performed separately on each loop:
   (a) Program slicing with respect to the current loop condition to simplify the program;
   (b) Value analysis to obtain variation domains with respect to loop variables;
   (c) Computation of loop bounds from the variation domains;
   (d) (For nested loops) Combination of results from inner and outer loops.

Note that step 1 can be performed by any loop reconstruction algorithm, and in most cases it is not considered as part of the analysis. Details about the algorithm are found in Section 2.2.5. Step 2 is performed separately on each loop for extra precision. For non-nested loops, the computed bounds at 2c are valid for the entire program. For nested loops, however, step 2d is necessary. Figure 4.1 illustrates the application of the two steps for the first loop in the program, while Figure 4.3 does the same for the second loop.

Loops bounded by their headers In Figure 4.1, loops are represented by gray rectangles. The loop bound analysis uses the loop header to bound all nodes in each loop. Instead of addressing each program point individually, it uses the property that *loop headers are executed at least as often as any other vertex in the loop*[2]. This property will be formally specified and proved in the formalization of the analysis (Section 4.4), but informally we can state this as a consequence of the fact that loops always begin executing in their headers, and must do so at each iteration. In the end, the method only computes one bound per loop and then propagates it to every other program point in the loop.

Program slicing To improve precision on each loop bound, we simplify the program by slicing it. Slicing with respect to the loop header results in a program which behaves the same way (i.e. has the same bounds), from the point of view of the header, but contains fewer statements. For instance, array accesses which do not affect the number of loop iterations are removed by program slicing, and otherwise we might not know how to deal with them. In principle, any program point in the loop might be used as slicing criterion, but for convenience we always choose the header, because it simplifies the proofs (since the header bounds the other vertices).

Our first loop has the vertex 4 as header, so it will be used as slicing criterion. Program slicing will simplify the example program P, obtaining another program P'_1 that is equivalent, with respect to the number of executions of vertex 4, to P. It means the loop bound in this simplified program is a valid bound for the original program P, so we can work on it instead of using P directly.

Value analysis With the sliced program P'_1, we perform a value analysis, more specifically an interval analysis, whose purpose is to obtain the domains of possible values for all program variables at the slicing criterion. In the example in Figure 4.1, the fourth column displays the abstract values inferred for each variable, at every program point in the sliced program. For the loop bound analysis, the only program point we are interested in is the loop header, where n has the singleton value 5 and i is between 0 and 5.

[2]Note that this does not include vertices belonging to loops which are nested in the current loop, because we bound each loop separately.

	Slice P_2' w.r.t. 2nd loop (at 6)	Value Analysis for P_2' (VA_{IN})	Interesting Variables (2nd loop)	Domain Sizes	Π(Sizes)	Π(Nested Loops)
1	`n = 5;`	n:?, i:?, j:?				
2	`i = 0;`	n:{5}, i:?, j:?				
4	`do {`	n:{5}, i:[0,5], j:?				
5	`  j = 0;`	n:{5}, i:[0,5], j:?				
6	`  do {`	n:{5}, i:[0,5], j:[0,5]	{j}	j: 6	6	$bounds_4 \times 6 = 36$
11	`  j++;`	n:{5}, i:[0,5], j:[0,5]				
12	`  } while (j≤n);`	n:{5}, i:[0,5], j:[1,6]				
13						
14	`  i++;`	n:{5}, i:[0,5], j:{6}				
15	`} while (i≤n);`	n:{5}, i:[1,6], j:{6}				
16	`return;`	n:{5}, i:{6}, j:{6}				

Figure 4.3: Loop bound analysis applied to the second (inner) loop of the example program. From left to right, the analysis steps are: value analysis, computation of *interesting variables* with respect to the second loop, and bound calculation (last 3 columns).

Bound calculation The information given by the value analysis allows us to quantify the maximum number of executions for each program point: since no repetition is allowed (due to our hypotheses of determinism and termination), we just count the number of combinations of possible values. Here are all the values for program variables n and i at program point 4: $\{5,0\}$, $\{5,1\}$, $\{5,2\}$, $\{5,3\}$, $\{5,4\}$ and $\{5,5\}$, for a total of 6 different combinations. Of course, we never actually enumerate such sets, we just multiply the sizes of the intervals: the singleton set $\{5\}$ for variable n has size 1, and the interval $[0,5]$ for variable i has size 6, therefore $1 \times 6 = 6$ is a safe bound for the outer loop.

Extra precision and nested loops The previous method is sufficient for loops which are not nested inside others, but imprecise otherwise. The actual method includes a few extra steps, omitted for simplicity, which we detail here when bounding the second loop. Figure 4.3 presents the program slice P_2', which is program P sliced with respect to to vertex 6, the header of the inner loop. The value analysis is performed just as before, obtaining intervals for n, i and j. This time, however, before computing the domain sizes, we compute a subset of the program variables that actually need to be considered in the product. Indeed, not all remaining program variables need to be taken into account: only variables that are *live*, *modified* and *used* inside the loop can contribute to the number of iterations[3]. Note that program slicing removes *statements*, but not variables. Also, due to control dependencies, some statements related to the outer loop remain in the program, even if they do not affect the inner loop. This "variable slicing" is complementary to program slicing and necessary for precision in nested loops. We call the remaining variables *interesting* and display those relative to the inner loop in Figure 4.3, next to its header.

Variables i and n are not considered interesting for the inner loop because they are neither used nor modified inside it. We can safely avoid including their variation domains in the product of the domain sizes. The inner loop can therefore be bounded to 6 iterations *per iteration of the outer loop*. Note that this bound is not absolute, but relative to the enclosing loop. To obtain global bounds, valid with respect to the *total* number of iterations of each program point, a final computation is needed. The multiplication of local bounds for each enclosing loop results in a valid global bound: $6 \times 6 = 36$ iterations, which is the final global bound for the inner loop header.

From Loop Bounds to WCET Estimations The final global loop bounds are converted into ILP constraints of the form $x_i \leq N$ and, in conjunction with the processor-behavior analysis and other ILP constraints, used to obtain WCET estimations. The structural constraints between

[3]In SWEET, this step is performed by what is called *invariant analysis*.

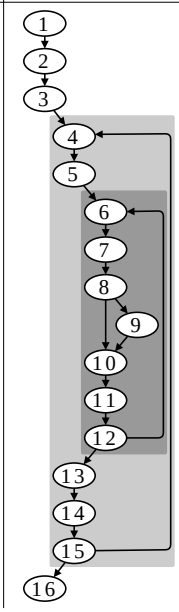
Program	CFG with Loops	Exact Bounds	Inferred Bounds
`n = 5;`		1	1
`i = 0;`		1	1
`w = 0.0;`		1	1
`do {`		6	6
`  j = 0;`		6	6
`  do {`		36	36
`    a[i][j] = i+1+j+1;`		36	36
`    if (i==j)`		36	36
`      a[i][j] *= 5.0;`		6	36
`    w += a[i][j];`		36	36
`    j++;`		36	36
`  } while (j≤n);`		36	36
`  b[i] = w;`		6	6
`  i++;`		6	6
`} while (i≤n);`		6	6
`return;`		1	1

Figure 4.4: Example program with its exact iteration bounds per program point and the bounds inferred from the results of our loop bound analysis. Only vertex 9 obtained an imprecise bound, due to the *if* condition.

vertices ensure that a single bound per loop is sufficient for the ILP to obtain a finite estimation. Figure 4.4 presents the exact bounds per program point and the ones inferred by the IPET after application of the loop bound constraints derived from our analysis. This is an example where the loop bound analysis succeeds: almost every program point is exactly bounded, except for vertex 9, due to the `if`. Extra precision can be obtained by improving the program slicing (e.g. finely computing dependencies), the value analysis (e.g. with pointer analysis or relational domains), or by adding other constraints to the ILP, obtained via different methods (e.g. specialized procedures for conditional branches).

4.2 Loop Bounds

Loop bounds are frequently found in the literature related to static analysis, compilation, WCET estimation and parallelization. They have applications related to program termination, loop optimization and parallelization, complexity analysis, etc. A precise definition of a *loop bound* is necessary to conduct the formalization of a method to estimate them. There are two kinds of loop bounds that are commonly referred to in the literature. The example program in Figure 4.6 serves as illustration for these bounds.

Local bounds (also called *relative* bounds) They correspond to the maximum number of iterations before exiting a loop. More precisely, let l_{in} be a program point inside the loop, and let l_{out} be a program point *before entering* the loop, e.g. a predecessor of the loop header. Local bounds correspond to the maximum number of iterations of l_{in} with respect to l_{out}.

In the literature, l_{in} is either not formally defined, or defined with incompatible semantics among different authors. In Figure 4.5 we present a program with a single loop to illustrate the ambiguity. It represents the C code `for(x=0;x<5;x++);`. The header of loop (2 3) is 2, its predecessor outside of the loop is 1. In this loop, l_{out} is vertex 1 and l_{in} is either vertex 2 or vertex 3, depending on the author. For instance, in SWEET [25] local bounds are applied to the header (i.e. vertex 2), while in Heptane [15] they are applied to vertices other than the loop entry (i.e. vertex 3). To distinguish between the two possible interpretations, we consider two kinds of local bounds:

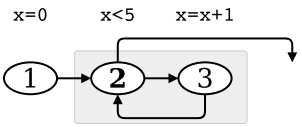


Figure 4.5: Program with a single loop, to illustrate the difference between the two kinds of local bounds. 2 is the loop header and 3 is the loop body. Its local *header* bound is 6, while its local *inner* bound is 5.

- *local header bounds*, defined as the maximum number of executions of the *header* with respect to the number of executions of the enclosing loop. This is equivalent to the sum of the executions of all header predecessors (l_{out} and the sources of back edges);
- *local inner bounds*, defined as the maximum number of times the loop *body* is executed (where *body* is the loop without its header).

The distinction between *header* and *inner* bounds is mostly useful for some kinds of loops, such as C-style `while` and `for` loops. For them, the loop condition is tested once for each loop iteration, and then one last time, when it evaluates to false and the loop is exited. For `do-while` loops, the *header* and *inner* bounds are equivalent, since the condition is at the end of the loop, and therefore executed as many times as the loop body. These relations no longer hold if there are extra loop exits,e.g. `break`s.

Local bounds are necessary to compute bounds for nested loops. They are also useful for manual annotations, because they are *context-free*: the user does not need to consider the entire execution context (enclosing loops, function calls) when inserting the annotation. For instance, in a simple loop such as `for(j=0;j<N;j++);`, the local bound is always N (or $N+1$, according to the loop bound semantics), so the user can immediately insert it as an annotation, even if the code is later transposed (e.g. inserted into another loop).

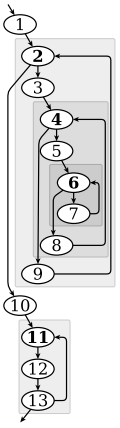
Global bounds (also called *absolute* bounds) They correspond to the maximum number of times a given program point can be reached during the entire execution of the program. The same semantic differences with respect to header/inner bounds are present here.

Global bounds are useful even in cases where local bounds are already available. In Figure 4.6, we have a program with triangular nested loops. The local bounds of the first loop are 5 (header) and 4 (inner). This loop is not nested inside any other, so its local and global bounds coincide. The second loop iterates a different number of times, depending on variable i: 0 iterations when $i = 1$, then 1 iteration, then 2, etc. Our bound annotations must be valid for all situations in a given program point. They do not vary between iterations, so in this case we consider the last iteration, when $i = 3$, which implies that, for $loop_2$, the header bound is 4 and the inner bound is 3. The global bounds in this case are different, however, because $loop_2$ is nested in $loop_1$. The global header bound of $loop_2$ is the sum $1+2+3+4 = 10$, the number of times the condition `j < i` is evaluated. The global inner bound is $0+1+2+3 = 6$. In the presence of triangular loops, global bounds are more precise than local bounds. For $loop_3$, we repeat the same reasoning. Finally, $loop_4$ is different, because it is a `do {...} while` loop. Inner and header bounds are equivalent in this kind of loop.

The estimation of global bounds for non-nested loops is identical to the estimation of local bounds. For nested loops, however, there are several different techniques, each with its own advantages and limitations. We present some of them in the next paragraph.

Estimating Nested Loop Bounds Nested loops pose interesting challenges regarding loop bounds. The simplest form of loop nesting consists of two loops whose number of iterations does not depend on one another. In this case, computing the product of the number of iterations of both loops constitutes a correct and precise[4] upper bound. For triangular loop nestings, this method also produces a correct bound, but it is no longer precise. For instance, in Figure 4.6, the product of the

[4]In this reasoning, we do not consider loops with several exits (such as `break`s). They lead to imprecise bounds in any kind of loops.




```
i = 0;
while (i < 4) { // loop1: (5, 4, 5, 4)
    j = 0;
    while (j < i) { // loop2: (4, 3, 10, 6)
        k = 0;
        while (k ≤ j) // loop3: (4, 3, 16, 10)
            k++;
        j++; }
    i++; }
i = 0;
do { // loop4: (3, 3, 3, 3)
    i++;
} while (i < 3);
```

Figure 4.6: Multi-loop program with annotated loop bounds, in the format (<*local header bounds*>, <*local inner bounds*>, <*global header bounds*>, <*global inner bounds*>).

local bounds of $loop_1$ and the local bounds of $loop_2$ is $5 \times 4 = 20$ for header bounds and $4 \times 3 = 12$ for inner bounds. Note that for loop nestings of three or more levels, instead of multiplying the local bounds of all enclosing loops, we simply multiply the *global* bounds of the immediately enclosing loop and the *local* bounds of the nested loop that we are bounding. This is computation method used by our loop bound algorithm, presented in the next section.

4.3 Loop Bounding Algorithm

From here on, we do not distinguish between header and inner bounds. Our algorithms and proofs operate on *header bounds*, which have a more direct formal justification than inner bounds. The latter require complex reasoning related to postdominators. When necessary, we can transform `for` and `while` loops into `do-while` loops by *inverting loops*[5], to eliminate issues related to imprecision.

Our method to compute global loop bounds uses a recursive algorithm that, starting from the beginning of the program, descends into each loop and estimates its bound, before recursively entering its nested loops. In the WCET literature, other techniques are found, a few of which are enumerated here:

- loop unrolling during symbolic execution: techniques based on symbolic program execution (such as SWEET [35]) can unroll loops during symbolic execution, using counters to compute global bounds as the total number of iterations (effectively bypassing the use of local bounds);
- polyhedral analysis: static evaluation of loop conditions allows some loops to be defined in terms of polyhedral (linear) relations between program variables (e.g., `x < y` and `y > 3` define a polytope in a 2-dimensional space). WCC [51] uses this technique, and then applies Ehrhart polynomial evaluation (that is, counting the number of integer points inside the polytope, each point corresponding to an execution of the loop) to obtain global bounds for statements inside the inner loop;
- recurrence relations: similarly to how algorithmic complexity analysis uses recurrence relations to estimate runtime complexity, WCET estimation tools such as r-TuBound [39] use them to estimate precise WCET bounds for nested loops. This requires simplifying the loops (e.g. by transforming a multi-path loop into a single-path one), but an advantage is that external solvers (e.g. based on computer algebra systems) can handle the computations of the closed forms of the relations.

[5] *Loop inversion* is the conversion of loops such as `while (cond) <body>` into `if (cond) {do <body> while (cond)}`.

All these techniques have in common the fact that obtaining tight global bounds for nested loops is not a trivial task. Indeed, much effort is directed towards methods to precisely compute such bounds, since overestimations can be very costly in terms of WCET.

Our formalized method for computing loop bounds defines a precise semantics for loop bounds in terms of execution traces of the source C program. We formalize local bounds, global bounds, and how to compute them by combining the formalized methods presented in the previous chapters.

Pseudocode

The pseudocode of the loop bound estimation (including invocation of program slicing and value analysis) is presented in Algorithms 1 and 2. By presenting them in detail, we can see some of the technical difficulties of the analysis and have a deep understanding of its workings. This is important to understand the difficulties that arise in the proof of correctness.

The loop bound estimation is performed by the `bounds` function, presented in an imperative programming style. Algorithm 1 presents the pseudocode of the recursive function that performs the actual bound computation, named `bounds_rec`, while Algorithm 2 presents the "external view" of the function, as seen by the user. Called `bounds`, this non-recursive function has a single input argument, the program P for which bounds will be computed. The result is a mapping from program points to bounds (positive integers, or $+\infty$ representing an unbounded value).

Algorithm 1: `bounds_rec`: recursive bounds computation

Input: P: program, lt: loop tree, acc: node $\to \mathbb{N}+\top$
Result: node $\to \mathbb{N}+\top$ (positive integer or $+\infty$)

foreach *(s: loop, ct: loop tree)* **in** $\texttt{children}(lt)$ **do**
  $h \leftarrow \texttt{header}(s)$
  $P' \leftarrow \texttt{slice}(P, h)$
  $val \leftarrow \texttt{value}(P')$ /* *val : node* $\times$ *variable* $\to$ *interval* */
  used-vars $\leftarrow \{x \in \texttt{vars}(P') | \exists l \in \texttt{nodes}(s), x \in \texttt{use}(l)\}$
  defd-vars $\leftarrow \{x \in \texttt{vars}(P') | \exists l \in \texttt{nodes}(s), x \in \texttt{def}(l)\}$
  interest-vars $\leftarrow \texttt{live}(P', h) \cap$ *used-vars* $\cap$ *defd-vars*
  $lb \leftarrow \prod_{x \in \textit{interest-vars}} |val(h, x)|$
  $gb \leftarrow lb \times acc(\texttt{parent}(h))$
  $acc \leftarrow acc[h \mapsto gb]$
  $acc \leftarrow \texttt{bounds\_rec}(P,ct,acc)$
end
return *acc*

Algorithm 2: `bounds`: non-recursive wrapper for `bounds_rec`

Input: P: program
Result: node $\to \mathbb{N}+\top$ (positive integer or $+\infty$)

$acc \leftarrow l_{\text{entry}} \mapsto 1$
$lt \leftarrow \texttt{reconstruct\_loops}(P)$
return `bounds_rec`*(P,lt,acc)*

In other words, `bounds` is merely a wrapper initializing these arguments:

- a partial result accumulator *acc*, which is a function mapping program points to execution bounds. Initially, as depicted in Algorithm 2, only the entry point has a mapping (it is associated to 1). After each calculation, a mapping is added for each loop header, in a recursive descent manner, until all loops have been mapped. The iteration can be performed either in breadth-first or depth-first order.
- a *loop tree lt*, representing the loop nesting structure of the program. For instance, Figure 4.7 depicts a program with several loops and its loop tree. Each node represents a loop, and

```
i = 0;
while (i < 4) { // loop1: (5, 4, 5, 4)
    j = 0;
    while (j < i) { // loop2: (4, 3, 10, 6)
        k = 0;
        while (k ≤ j) // loop3: (4, 3, 16, 10)
            k++;
        j++; }
    i++; }
i = 0;
do { // loop4: (3, 3, 3, 3)
    i++;
} while (i < 3);
```

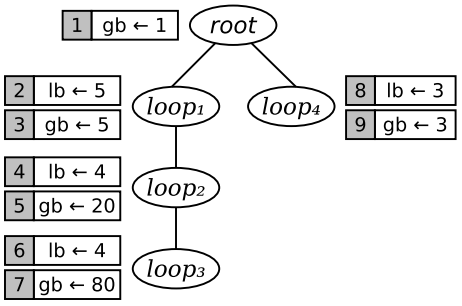


Figure 4.7: Program (the same from Figure 4.6) with its associated loop tree, decorated with estimated loop bounds (*lb* for local bounds, *gb* for global bounds). The numbers in gray indicate a possible iteration order during computation.

each child is a nested loop. Nodes can have any number of children, which are themselves loop trees. The root of the tree is not a loop *per se*: it is the *base scope*, whose header is the program entry point;

We suppose the loop tree can be obtained using a loop reconstruction algorithm (which we called `reconstruct_loops`). Note that we consider the program P to have a normalized CFG, that is, its entry point l_{entry} is unique and has no predecessors, which ensures it will be executed only once. Also note that the mapping function *acc* contains not only positive integers, but also the $\top$ value, used when a loop cannot be bounded.

We now describe, line by line, the workings of `bounds_rec` (Algorithm 1):

- **line 1**: our algorithm iterates through each child loop of *lt*, if any, to compute bounds for this loop. s is the loop itself, and *ct* is the list of its children (which is empty if s has no inner loops). If *lt* itself has no children, that means the loop containing it has already been bounded in a previous step, and therefore nothing needs to be done.
- **line 2**: obtains the loop header h for loop s.
- **line 3**: computes a program slice using h as slicing criterion. The resulting slice P' is a simpler version of P, which improves the precision[6] of the following steps (see Figure B.1 in Appendix B for a concrete example of improved precision due to slicing).
- **line 4**: value analysis computation on the program slice P'. The result is a mapping from pairs *(program point, variable)* to intervals. The only values we use are those associated to program point h. The value analysis is performed *after* program slicing to benefit from the extra precision it confers.
- **lines 5-7**: compute the *interesting variables* (described in Section 4.1), which are merely the intersection of variables used and defined inside the loop, and live at the loop header. They include all the necessary variables to obtain correct loop bounds.
- **line 8**: computes the local bounds *lb* for loop s. First, we retrieve the result of the value analysis for program point h. Then, for each interesting variable x, we obtain the interval of its possible values at h, and measure the length of this interval ($|val(h, x)|$). The local bounds for the loop are the product of the size (length) of all intervals.

[6]Program slicing also usually improves the execution time of the analysis, since the program slice is smaller than the original program, but here our main concern is precision.

- **line 9**: the global bounds for loop s are estimated via a product of the *local* bounds at h and the *global* bounds at the parent of h (whose global bounds have already been computed).
- **lines 10-11**: update the *acc* structure with the loop bounds for s, then recursively call `bound'` on the children of s. The recursive call returns *acc* with extra mappings. Repeat the process for each child of lt.
- **line 13**: returns *acc* (unmodified, if the current loop tree was empty, or with added mappings otherwise) as the final result of the function. Eventually returns to the original `bounds` call, with *acc* completely filled.

Note that the final returned *acc* mapping only contains associations for loop headers. We can easily extend these associations to every program point l, by returning $acc(\texttt{header}(l))$. This is a safe approximation, due to the fact that *loop headers are executed at least as often as any other node in the same loop*[7].

We apply the loop bound analysis as an intermediate step during program compilation. The result of this process is that, when we compile a program P_C into a program P_{Asm}, we additionally obtain a mapping from program points to execution bounds, which we call $\texttt{bounds}_P$ (the result of applying the `bounds` function to a program). This mapping is used in the correctness theorem of the loop bound analysis, presented in Section 4.4.

4.4 Correctness

Figure 4.8 presents the general architecture of the loop bound estimation tool, highlighting in bold the parts described in this chapter. The next chapters describe the implementation and proof of the value analysis and program slicing.

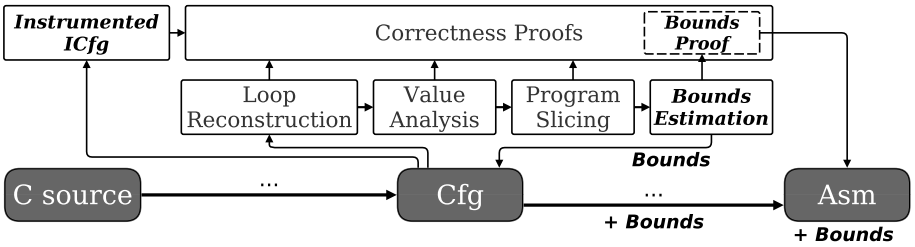


Figure 4.8: Architecture of the loop bound analysis implemented in CompCert, with the items related to the proof of the loop bound estimation highlighted (in bold and italics).

We describe here the correctness proof formalized in Coq, with some extra presentation information to help understand the main aspects of the proof, in particular the semantic justifications that ensure the result is correct. This is a top-down presentation: we begin with the main correctness theorem that we want to prove, then in Section 4.4.1 we present the instrumented semantics used in the proof. In Section 4.4.2, we describe an intermediate correctness theorem, whose result is used to prove the former. The proof of this theorem is described in Section 4.4.3, where we present the three lemmas used in the proof.

The main correctness theorem of the loop bound estimation relates *execution bounds* to *events* in the event trace of a program (this trace has been described in Section 2.2.2). Since CompCert's semantics ensures the preservation of such events during compilation, we can directly associate *annotations* in the source program to *event occurrences* in the assembly program. For instance, consider the following program:

[7] This property is stated as a lemma in the proof of correctness, and further detailed in Section 4.4.

```
1  int main() {
2    int i = 0;
3    printf("begin");
4    do {
5      __annot("LOOP");
6      i++;
7    } while (i < 3);
8    __annot("end");
9    return 0;
10 }
```

The execution trace of this program, given by the CompCert semantics, is the following *event list*, which we name `tr`:

`tr = [extcall printf(“begin”), annotation “LOOP”, annotation “LOOP”, annotation “LOOP”, annotation “end”].`

The notation `extcall printf` denotes that `printf` is an *external function call*, which is an observable event according to CompCert's semantics. *Annotations* are another kind of observable event. Both kinds of events contain *labels* (character strings) which are also part of the observable trace. For instance, annotations containing the `“LOOP”` string are distinct from those containing the `“end”` string. Annotations do not change the semantics of a program. They can be automatically inserted at each loop, for instance, or in other points of interest: function entries, `if` branches, etc.

For program points which generate observable events, the number of generated events in the execution trace is exactly the number of executions of that program point. Since CompCert's correctness theorem (Theorem 1) ensures the preservation of observable behaviors all throughout the compilation, this property is valid for any language, from C down to the assembly code. We thus leverage CompCert's trace preservation theorem, which enables us to perform an analysis at any level in the compiler (in our case, ICfg) and obtain results that are valid for the compiled code. This leads us to the main correctness theorem of the loop bound estimation, Theorem 2, which states that the result of the loop bounding algorithm is a correct bound of the number of occurrences of an event in the execution trace of the program[8].

In Theorem 2, we use the notation $\#\texttt{tr}_{\downarrow\texttt{al}}$ (where `tr` is an event trace and `al` an annotation label) to represent the number of occurrences of the event attached to `al` in `tr`. For instance, in our previous event list, we have $\#\texttt{tr}_{\downarrow\texttt{annotation “LOOP”}} = 3$. Note that annotation labels are a language-independent concept, unlike program points, which are specific to each intermediate language. For this reason, instead of directly using the result of the `bounds` function defined in Section 4.3, which is specific to the ICfg language, we use $\texttt{bounds}_P$, a mapping from annotation labels to occurrence counters. $\texttt{bounds}_P$ is computed, for each annotation label, as the sum of the execution counters $\texttt{bounds}(P)(l)$ for every program point l associated to that label—which corresponds to the maximum number of occurrences in the events trace. Usually, each annotation label is uniquely defined in a program, but transformations such as loop unrolling may duplicate annotations.

Theorem 2 (Start-to-end correctness)**.**
Let P_C be a source C program, free of runtime errors.
Let P_{Asm} be the result of the compilation of P_C, and $\texttt{bounds}_P$ the result of the loop bound analysis performed during compilation.
Then, for any finite execution of P_{Asm} that produces a trace of events `tr`*,*
and for any annotation label `al`*, we have:*

$$\#\texttt{tr}_{\downarrow\texttt{al}} \leq \texttt{bounds}_P[\texttt{al}]$$

[Coq Proof][9]

With Theorem 2, it is sufficient to define an annotation label at each loop header to obtain the necessary bounds for WCET estimation. The ILP generator for IPET will later use these bounds,

[8] We consider here a program free of runtime errors, that is, a program whose executions do not *go wrong*. Going-wrong behaviors are defined in Section 2.2.2.

[9] This notation indicates a hyperlink in the electronic version to the proof of the theorem in the online development.

associating the annotation labels to the assembly program points where they are defined, to bound their execution counters.

To prove Theorem 2, we need to establish a relation between $\texttt{bounds}_P$ and CompCert's event trace. To do this, we introduce an ICfg semantics instrumented with execution counters per program point. These counters will then be used to establish relations between program points and number of executions, and these relations will in turn be used to prove that the values computed by $\texttt{bounds}_P$ are indeed execution bounds.

4.4.1 Counter Instrumentation for ICfg

We instrument the ICfg language with *execution counters* at each program point. We also call them *global* execution counters.

These counters change the semantics of the ICfg language. Each state now has an extra element, $c : \mathcal{PP} \to \mathcal{N}$ which maps program points to natural integers corresponding to the number of executions of these program points during program execution. A semantic state is therefore a quadruple (l, R, M, c) where the first three elements (program points, registers and memory contents) are left unchanged with respect to the original semantics and the last element, c, is the mapping of the execution counters.

The new step relation, noted $\to_i$, is defined as follows: an instrumented step from (l, R, M, c) to (l', R', M', c') is a non-instrumented step from (l, R, M) to (l', R', M') plus an increment to the counter $c(l)$. Initially, the counter of each program point is associated to zero. This increment corresponds to the fact that "l has been iterated". Formally, this becomes:

$$\frac{(l, R, M) \to (l', R', M') \qquad c' = c[l \leftarrow c(l) + 1]}{(l, R, M, c) \to_i (l', R', M', c')}$$

This semantics corresponds to the intuitive notion of *execution count* for a given program point, and it conforms to the usage in WCET-related flow analyses, such as IPET, where each program point has an associated integer variable which counts this number of executions.

This instrumentation is sufficient for defining the final correctness theorem of the loop bound analysis, but it is not enough to actually develop the intermediate proofs which lead to it. For this, we use another set of counters, called *local execution counters*, which are related to each loop in the program. The only difference between local and global counters is that all local counters of a loop are reset to zero when the loop is exited[10]. This produces counters which are *relative to each loop entry*, and this information will be useful for the loop bound analysis. We write $l'' \in \textit{exited-loops}(l, l')$ when l'' is inside a loop that has just been exited when stepping from l to l'. For instance, in the program in Figure 2.8, we have $\textit{exited-loops}(2, 3) = \emptyset$ (no loops are exited when traversing from 2 to 3) and $\textit{exited-loops}(9, 11) = \{5, 6, 7, 8, 9, 10\}$.

The final formal semantics of the step relation of ICfg (defined in Section 2.2.7) instrumented with both sets of counters ($\to_{\text{cs}}$) is the following (we denote the global counters by c_{glob} and the local counters by c_{loc}):

$$\frac{\begin{array}{c}(l, R, M) \to (l', R', M') \\ c'_{\text{glob}} = c_{\text{glob}}[l \leftarrow c_{\text{glob}}(l) + 1] \qquad c'_{\text{loc}} = \textit{reset-exited}(c_{\text{loc}}[l \leftarrow c_{\text{loc}}(l) + 1], l, l')\end{array}}{(l, R, M, c_{\text{glob}}, c_{\text{loc}}) \to_{\text{cs}} (l', R', M', c'_{\text{glob}}, c'_{\text{loc}})}$$

Let $c_{\text{reset}} = \textit{reset-exited}(c_{\text{loc}}, l, l')$. Then, for any l'',

$$c_{\text{reset}}(l'') = \begin{cases} 0 & \text{if } l'' \in \textit{exited-loops}(l, l') \\ c_{\text{loc}}(l'') & \text{otherwise.} \end{cases}$$

This is the ICfg instrumented semantics used in the proof of the loop bound analysis. It is equivalent to the original ICfg semantics with erased execution counters.

[10] When considering the maximum values of execution counters, resetting them at loop entries is equivalent to doing it at loop exits. However, the proof of program slicing applied to the loop bound analysis (Section 5.4.1) is easier when resetting counters at loop exits.

4.4.2 Bounds Correctness with Instrumented Semantics

Using the instrumented semantics, and the fact that annotations generate events in the trace, we link $\#\mathtt{tr}_{\downarrow \mathtt{al}}$ to the global counters c_{glob} in the semantics. We then relate c_{glob} to $\mathtt{bounds}_P$ via Theorem 3 below, which states that for any program point l of P, the bound estimation at l is a correct estimation of the global execution counter at this point. We denote the terminating execution of program P with counters c as $P \Downarrow_t c$.

Theorem 3 (Semantic bounds correctness)**.**
Let P be an ICfg program such that $P \Downarrow_t c_{glob}$, and let l be a program point of P.
Then, we have:

$$c_{glob}(l) \;\leq\; \mathtt{bounds}(P)(l)$$

[Coq Proof]

In our previous example (Figure 4.9), when execution terminates at line **9**, the semantics gives us the following counters for the program:

$c_{\text{glob}} = \{\mathbf{2} \mapsto 1, \mathbf{3} \mapsto 1, \mathbf{4} \mapsto 3, \mathbf{5} \mapsto 3, \mathbf{6} \mapsto 3, \mathbf{7} \mapsto 3, \mathbf{8} \mapsto 3, \mathbf{9} \mapsto 1\}$

In this simple program, this is the exact result returned by the `bounds` function at each program point:

$\mathtt{bounds}_P(\mathbf{2}) = 1$, $\mathtt{bounds}_P(\mathbf{3}) = 1$, $\mathtt{bounds}_P(\mathbf{4}) = 3$, ...

4.4.3 Semantic Correctness Proof

The proof of Theorem 3 uses three lemmas. We present each of them in turn, and by composing them we can prove that the algorithm presented in Section 4.3 produces correct bounds.

The first lemma, *header counters dominate body counters*, is a consequence of the constraints imposed by the loop reconstruction algorithm. It produces structured loops having a single entry point (the header) and all back edges pointing to this node. As a consequence, no other node in the same loop[11] can be executed more often than the header. This lemma is stated below. Note that we do not specify which counters c refers to, because the lemma applies to both local and global counters.

Lemma 1 (Header counters dominate other counters in the loop)**.**
For any reachable state $\sigma \in \mathtt{reach}(P)$ and any vertex l of P, we have:

$$\sigma.c(l) \;\leq\; \sigma.c(\mathtt{header}(\mathtt{loop}(l)))$$

[Coq Proof]

The proof of Lemma 1, detailed below, depends on an inductive property related to execution traces. The most difficult part of the proof, in this lemma and in the next ones, is to find a correct invariant that is strong enough for the proof.

Proof of Lemma 1. We establish this property by proving, by induction on finite execution traces, that for any vertex l, distinct from its header $l_h = \mathtt{header}(\mathtt{loop}(l))$, and for any partial finite execution trace $\xi = \sigma_0, \sigma_1, \ldots, \sigma_n$, one of three following conditions holds:

- either the expected inequality holds strictly: $\sigma.c(l) < \sigma.c(l_h)$,
- or l has not been reached yet: $\sigma.c(l) = 0$,
- or $\sigma.c(l) = \sigma.c(l_h)$ but there exists $k \in [0, n-1]$ such that σ_k is at the program point l and all states $\sigma_{k+1}, \ldots, \sigma_{n-1}$ did not reach the header l_h.

The last condition implies that we cannot reach vertex l in σ_n: it would build a cycle from l to l that does not contain l_h, and this is forbidden by the property *cycles must include headers*, mentioned in Section 2.2.5 and obtained by the loop reconstruction validator.

□

[11] Nodes in inner loops maintain this relation with respect to their respective headers.

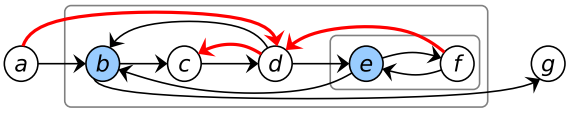


Figure 4.9: Example CFG to illustrate Lemma 1. The validity of the lemma depends on the fact that the red edges are not allowed by the loop reconstruction algorithm.

Despite the difficulty of finding a sufficiently strong inductive invariant, this proof is fairly simple to explain informally. We use the CFG in Figure 4.9, with added red arcs to illustrate situations which would invalidate the lemma, such as unstructured loops. For instance, the arc $a \to d$ is invalid, because all loop entries must include the loop header, which is b in this case. The loop reconstruction phase rejects such CFGs.

The arc $d \to c$ is also forbidden, since it is a back edge (from a right-most vertex to a left-most one) and its destination is not a loop header. In practice, this never happens because, in this case, the loop reconstruction algorithm would produce an extra loop containing both c and d, and then the back edge would be valid. The same would happen if there were an edge $f \to d$. In this case, d would be header of a loop containing itself, e and f.

Now that the shape of our CFG is established, we can verify that the inductive invariant is indeed an invariant. Initially, all counters are zero. Then, since b is the only loop entry, its counter is incremented. After stepping to c, c's counter is not greater than b's. The only way to reach c again is through b, so before we can increment its counter again, we will have already incremented b's counter. Note that f's counter may be larger than b's, but that's not a problem, since f's header is e. At each execution, we may skip some vertices (not incrementing their counters), but never the loop header, therefore its counter is never smaller than the others.

This lemma allows us to reuse the bounds of a header, $\texttt{header}(\texttt{loop}(l))$, as bounds for all vertices in the same loop: $\texttt{bounds}(P)(l) = \texttt{bounds}(P)(\texttt{header}(\texttt{loop}(l)))$.

The next lemma (Lemma 2), *variable domain sizes bound local counters*, applies the result of the value analysis and the interesting variables computation to obtain bounds for local execution counters. We use $|[a,b]| = b - a + 1$ to denote the size of interval $[a,b]$.

Also, we use P' (the program slice) instead of P as a reminder that the value analysis is applied on the program sliced with respect to the loop header. The slice contains fewer interesting variables and smaller intervals, which improves the precision of the result without impacting correctness. Note, however, that the proof still works even if we use the original program.

Lemma 2 (Domain sizes of interesting variables bound local counters)**.**
For any reachable state $\sigma \in \texttt{reach}(P')$*, we have:*

$$\sigma.c_{loc}(l_h) \;\leq \prod_{x \in \texttt{live}(l_h) \cap \texttt{use}(\texttt{loop}(l_h)) \cap \texttt{def}(\texttt{loop}(l_h))} |\texttt{value}(P')(l_h)(x)|$$

[Coq Proof]

We implicitly extend the notations `use` and `def` to sets of program points. They denote the union of the variables used/modified in any program point belonging to the set. The proof of Lemma 2, which relies on the hypothesis that the program terminates, is intuitively simple but quite complex to formalize in a proof assistant. Figure 4.10 presents example program P_L, used to illustrate parts of the proof. We write a state in the trace of the program as $(l : b, i, s)$, where l is the state's current program point and (b, i, s) is the projection of the values of variables b, i and s.

Proof of Lemma 2. Given a vertex l, we use $IV(l)$ to denote the set $\texttt{live}(l) \cap \texttt{use}(\texttt{loop}(l)) \cap \texttt{def}(\texttt{loop}(l))$ of interesting variables at l. In Figure 4.10, $IV(4) = \{b, i\}$ (since s is not *used* inside the loop).

We first prove that, if there exists an execution trace $\xi = \xi_1 \cdot \sigma_1 \cdot \xi_2 \cdot \sigma_2$ such that $\sigma_1.l = \sigma_2.l$ and both states σ_1 and σ_2 match pointwise on each variable of $IV(l)$, then we can build a valid execution trace of arbitrary large size by repeating $\xi_2 \cdot \sigma_2$ indefinitely: $\xi_1 \cdot \sigma_1 \cdot (\xi_2 \cdot \sigma_2)^N$. This is an intended consequence of the definition of interesting variables: we can safely ignore all *un*interesting

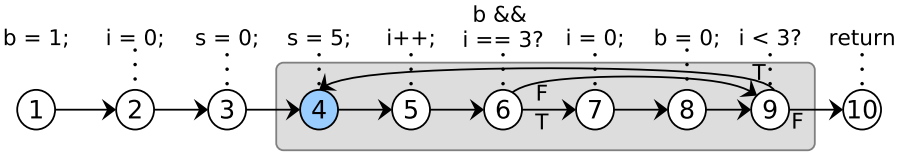


Figure 4.10: Example program P_L used to illustrate the proof of Lemma 2. The loop iterates 6 times in total. This kind of loop cannot be bounded just by looking at the variables involved in the loop exit condition (vertex 9).

variables when considering branch decisions, since they will either have a constant value inside the loop, or they will not influence at all (neither directly nor indirectly) the loop decision. We combine it with the hypothesis about determinism, which ensures that multiple evaluations of the condition with the same input will result in the same output. Combined, they mean that if we reach a same state $(l : b, i)$ (ignoring the value of s) twice in the trace, then we reach it infinitely many times. Since we assumed that P' terminates[12], we obtain a contradiction.

Now, any execution trace ξ reaching at least once the loop header of l, which we denote by l_h, can be divided into $\xi = \xi_1 \cdot \sigma_1 \cdot \xi_2 \cdot \sigma_2$ where σ_1 is the last state in the execution that enters the loop nesting of l_h. The counter $\sigma_2.c_{\text{loc}}(l_h)$ is equal to the length of the sub-trace ξ^h that we obtain by projecting $\sigma_1 \cdot \xi_2 \cdot \sigma_2$ on the states that are at vertex l_h.

For instance, in Figure 4.10, considering the execution trace from the beginning of the program until just before exiting the loop, we have $\sigma_1 = (4 : 1, 0, 0)$ (when entering the loop) and $\sigma_2 = (9 : 0, 3, 5)$ (just before the final test which will exit the loop). $c_{\text{loc}}(4)$ is 6 at this point, which corresponds to the length of the trace ξ projected on vertex 4: $\{(4 : 1, 0, 0), (4 : 1, 1, 5), (4 : 1, 2, 5), (4 : 0, 0, 5), (4 : 0, 1, 5), (4 : 0, 2, 5)\}$.

Each state in ξ^h can be turned into a n-tuple (where $n = |IV(l_h)|$) containing the value of each variable of $IV(l_h)$ in this state. Mapping this transformation on ξ^h, we obtain a list of size $\sigma_2.c_{loc}(l_h)$. This list contains distinct n-tuples thanks to the *Reductio ad absurdum* we made early in this proof. By soundness of the value analysis, each n-tuple belongs to the direct product of the interval $\texttt{value}(P')(l_h)$. In our example in Figure 4.10, this list will contain the 6 pairs $(1, 0)$, $(1, 1)$, $(1, 2)$, $(0, 0)$, $(0, 1)$ and $(0, 2)$.

We prove that there exists a list of size $\prod_{x \in IV} |\texttt{value}(P')(l_h)(x)|$ (that is, the product of size of the domains of all interesting variables IV at the loop header l_h in the program slice P') containing all the possible n-uples of this direct product and conclude our proof by a pigeonhole argument. In our example, this is obtained by computing $|[0, 1]| \times |[0, 2]|$ (sizes of the intervals of variables b and i, respectively). An enumeration of all pairs of elements in these intervals results precisely in the list of 6 pairs mentioned previously. Inserting an extra element in the list would amount to repeating one of the previous pairs, and such a repetition would imply non-termination, contradicting our hypothesis.

□

At this point, we have already obtained correct local bounds for our loops. However, the semantics of local bounds is unintuitive, since it is related to *resettable* counters, whose behavior is related to an instrumented CFG semantics and loop entries, which depend on the loop reconstruction algorithm... In sum, such local bounds cannot be easily transposed to the assembly level, where the WCET estimation will likely take place. Global bounds are better suited for this. Therefore, to transpose one type into the other and relate them to Theorem 2, we use Lemma 3, which details the *conversion of local bounds into global bounds*.

Intuitively, we can reason as follows: let N be the local bound for the header l_h of a loop s. As defined in Section 4.2, a local bound is a bound *relative to the immediately enclosing loop*, that is, relative to the bounds of $\texttt{parent}(l_h)$. Let us call this loop p. Each time we *enter* loop s from loop p, we can only iterate l_h up to N times before leaving loop s. But after iterating the header of p once more, we again may iterate l_h up to N times. This amounts to iterating l_h a total of

[12]It is not always the case that program slicing preserves termination, so our slicing must ensure the following implication: if P terminates, then P' terminates. This is demonstrated in Chapter 5.

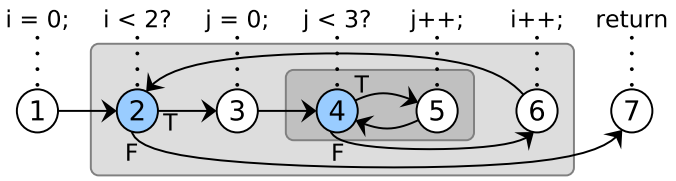


Figure 4.11: Example program P_G, consisting of two nested loops, used to illustrate Lemma 3.

$N + N + \cdots + N = \mathit{bounds}(\texttt{header}(p)) \times N$. This is the total, global, number of times we can pass by l_h during the entire program execution. And $\mathit{bounds}(\texttt{header}(p))$ corresponds to the *global* bounds of the parent scope p. Finally, we have established bounds which are independent of any local notions.

Lemma 3 (From local to global bounds)**.**
Let l_h be a loop header and l_p the header of its parent loop, i.e. $l_p = \texttt{header}(\texttt{parent}(\texttt{loop}(l_h)))$. Let N be a bound for the local counter of l_h: $\forall \sigma \in \texttt{reach}(P), \sigma.c_{loc}(l_h) \leq N$. Then, we have:

$$\forall \sigma \in \texttt{reach}(P), \sigma.c_{glob}(l_h) \leq N \times \sigma.c_{glob}(l_p)$$

[Coq Proof]

The proof of this lemma follows the same reasoning presented previously, but the formal arguments are much more complex, requiring a strong inductive property and several auxiliary properties related to the loop structure. To help visualize the property, we use the simple program P_G in Figure 4.11. In this figure, we consider the header node 4 and its parent 2. We can compute the entire execution trace of program P_G, projected here on its program points (without variable values):

$$1,2,3,\underbrace{\underbrace{4,5}_{L_j},\underbrace{4,5}_{L_j},\underbrace{4,5}_{L_j},4}_{L_i},6,2,3,\underbrace{\underbrace{4,5}_{L_j},\underbrace{4,5}_{L_j},\underbrace{4,5}_{L_j},4}_{L_i},6,2,7$$

We divide this trace in "complete executions of the outer loop" (indicated as L_i) and "complete executions of the inner loop" (indicated as L_j). The proof uses a similar structure, dealing with subtraces divided at each passage through the parent header 2. These subtraces impose a limit on the number of times the header node 4 may be traversed: we know, due to Lemma 2, that whenever we enter the loop at 4, we must exit at the latest after N executions, where N is the local bound of loop 4. After exiting the loop of vertex 4, if we want to iterate it again, we must do it in one of two ways:

- we exit the parent loop (at header 2) and return to it later somehow;
- we do not exit the loop of vertex 2, but we take a back edge to return to its header, and from there we can reach vertex 4 again.

In both cases, we are forced to execute vertex 2 at least once. So, for each execution of the parent loop, we can iterate the inner loop up to N times. If the number of executions of vertex 2 is bounded by $K = c_{\text{glob}}(l_p)$, then the number of executions of vertex 4 is bounded by $K \times N$.

Proof of Lemma 3. We first consider execution traces of the form $\xi = \xi_0 \cdot \sigma_p \cdot \xi_1 \cdot \sigma$ such that σ_p is a state at vertex l_p and no state in trace ξ_1 has reached l_p again.

On such traces we show that $\sigma.c_{\text{glob}}(l_h) - \sigma_p.c_{\text{glob}}(l_h) \leq N$ holds. Unfortunately, this property is not inductive. We strengthen it into a disjunction where:

- either $\sigma.c_{\text{glob}}(l_h) = \sigma_p.c_{\text{glob}}(l_h)$ and no state in ξ_1 has reached l_h yet,
- or $\sigma.c_{\text{glob}}(l_h) = \sigma_p.c_{\text{glob}}(l_h) + \sigma.c_{\text{loc}}(l_h)$ and the state σ is currently inside the loop of l_h,
- or σ is currently outside the loop of l_h, $\sigma.c_{\text{glob}}(l_h) - \sigma_p.c_{\text{glob}}(l_h) \leq N$ and l_h has been reached during ξ_1.

We prove this disjunction by induction on the execution trace ξ_1.

To conclude this proof, we consider an arbitrary trace $\xi = \sigma_0 \cdots \sigma$ and we divide it into $K + 1 = 1 + \sigma.c_{\text{glob}}(l_p)$ subtraces $\xi = \xi_0 \cdot \xi_1 \cdots \xi_K$ such that $\forall i \in [1, K], \xi_i$ starts with a state σ_i at point l_p and then never reaches it again. We note $\sigma_{K+1} = \sigma$. We then express $\sigma.c_{\text{glob}}(l_h)$ as

$$\sigma.c_{\text{glob}}(l_h) = \sigma_0.c_{\text{glob}}(l_h) + \sum_{k=0}^{K-1} (\sigma_{k+1}.c_{\text{glob}}(l_h) - \sigma_k.c_{\text{glob}}(l_h))$$

In the initial state σ_0, every counter is null and each element in the sum is bounded by N. Thus, we conclude that $\sigma.c_{\text{loc}}(l_h) \leq K \times N$ and finish the proof since $K = \sigma.c_{glob}(l_p)$. □

We compose the three previous lemmas and prove Theorem 3, by unfolding the definition of the `bounds` function (Algorithm 2) and applying, at each step, the appropriate lemma. By matching the counters given by the semantics with the bounds given by the algorithm, we ensure the correctness of the loop bound estimation.

4.5 Conclusion

The formalized loop bound estimation method presented here started with the formal definition of loop bounds, both local and global, and an instrumented ICfg semantics—with execution counters—used to formally reason about these bounds. Afterwards, to formally verify the algorithm, we programmed it in Coq and proved that the result of its computation always corresponds to an upper bound of the actual execution counters defined in the semantics.

Although informally the reasoning about loop bounds seems "trivial", some considerable proof effort is needed to provide the necessary semantic guarantees. Starting from the pigeonhole principle and going all the way up to the recursive computation of nested loop bounds requires a systematic approach, with a decomposition in several intermediate steps. In the end, we obtained guarantees about its correctness, but nothing is said about its precision. To analyze and evaluate this aspect, we performed an experimental evaluation which is described in Section 7.3.

Chapter 5

Program Slicing

Program slicing is a technique to simplify a program with respect to a criterion, that is, a variable at a given program point. It consists in the removal of statements which have no influence on the criterion. The statements thus removed are said to have been *sliced away*. The result of this process is another program, a simplified version of the original one, and it is called a *slice*.

By itself, program slicing can be useful for debugging and program understanding. It is often used in conjunction with other transformations and analyses, as a preprocessing step. An important property of program slicing is that *the program slice behaves as the original program*, with respect to the slicing criterion.

Since its original definition by [68], where it is seen as a tool to help debugging, program slicing has been the subject of decades of extensive work that have spawned several variants, such as *dynamic slicing*, *forward slicing*, *amorphous slicing*, etc. [66] presents an overview of program slicing, detailing several techniques to perform static and dynamic slices. [32] succinctly describes the differences among these variants. Here, we consider the original form of program slicing, where it can be seen, from a static analysis point of view, as a method that computes a safe over-approximation of all statements possibly influencing a given slicing criterion.

In this chapter, we state some terminology related to program slices in Section 5.1. Then, we present the concepts related to program slicing techniques, (such as data dependencies and control dependencies), as well as the architecture and implementation of our program slicer in Section 5.2. This allows us to obtain untrusted program slices. We then define the correctness of our slices with respect to the preservation of values for relevant variables in Section 5.3. We also present our proof strategy combining *a posteriori* validation and proofs by simulation. We modify the program slicer to include a checker, allowing it to produce verified program slices. Afterwards, we present the application of program slicing to the loop bound estimation (presented in Chapter 4). Section 5.4 presents the correctness theorem and proofs for this application, as well as the modifications to the algorithm that enable it to generate verified slices for the loop bound analysis. Finally, we conclude in Section 5.5, where we also express some remarks concerning limitations of our slicing and indications for further work.

5.1 Program Slicing Terminology

In this section, we define the main concepts related to program slicing, which will be used throughout the chapter. To aid in understanding some of the concepts, we present an example C program[1] with different slices in Figure 5.1. This program is the same one presented in Figure 4.1, containing two nested loops. P_1', which is the slice of P_1 related to the outer loop and vertex 4, removes almost everything in the program. The slice P_2', related to the inner loop and vertex 6, only removes a few statements. Finally, Figure 5.1 includes a variant of P_2' in the last column, named P_2''. This slice is also related to vertex 6, but it is a so-called *CFG-preserving* slice: it preserves the control flow

[1] Our algorithms and proofs have been implemented on the Cfg language, but the examples are presented in C to improve readability. Defined and used variables are the same in C and Cfg (except for C memory variables, replaced by *Mem* in our analyses). Our use of `if` and `while` control statements ensures that program points are also preserved.

Program P	CFG	Slice P_1' w.r.t. 1st loop (at 4)	Slice P_2' w.r.t. 2nd loop (at 6)	CFG-Preserving Slice P_2'' w.r.t 2nd loop (at 6)
`n = 5;`	1	`n = 5;`	`n = 5;`	`n = 5;`
`i = 0;`	2	`i = 0;`	`i = 0;`	`i = 0;`
`w = 0.0;`	3			`skip;`
`do {`	4	`do {`	`do {`	`do {`
`  j = 0;`	5		`  j = 0;`	`  j = 0;`
`  do {`	6		`  do {`	`  do {`
`    a[i][j] = i+1+j+1;`	7			`    skip;`
`    if (i==j)`	8			`    if (false)`
`      a[i][j] *= 5.0;`	9			`      skip;`
`    w += a[i][j];`	10			`    skip;`
`    j++;`	11		`    j++;`	`    j++;`
`  } while (j≤n);`	12		`  } while (j≤n);`	`  } while (j≤n);`
`  b[i] = w;`	13			`  skip;`
`  i++;`	14	`  i++;`	`  i++;`	`  i++;`
`} while (i≤n);`	15	`} while (i≤n);`	`} while (i≤n);`	`} while (i≤n);`
`return;`	16			`return;`

Figure 5.1: Example program P with its CFG, its slice (P_1') with respect to vertex 4, slice (P_2') with respect to vertex 6, and slice (P_2''), a variant of (P_2') that preserves the CFG.

structure of the original program. Slices P_2' and P_2'' are equivalent in terms of program simplification, but the CFG-preserving slice offers an advantage: its transformation is simpler to prove correct. For this reason, our slicing algorithm produces CFG-preserving slices.

A *program slice* (also called simply a *slice*) is a program, a simplified version of the program which has been sliced. A slice is defined with respect to a *slicing criterion*. Traditionally, a criterion comprises either a program point, or a variable at a given program point. In our case, we consider a program point, which we refer to as l_s, and indirectly all variables *used* (as defined by the *use* function in Section 2.2.4) at this point. For instance, in Figure 5.1, the slicing criterion for slice P_1' is program point 4, and for slices P_2' and P_2'' it is 6.

We define a *slice set* (denoted as $\texttt{SL}(l_s)$ for a given slicing criterion l_s) as the set of program points that are preserved by the slicing. In Figure 5.1, the slice set for slice P_1' is composed of all program points which remain unchanged in the slice: $\texttt{SL}(4) = \{1, 2, 4, 14, 15\}$. Note that, in a CFG-preserving slice such as P_2'', the slice set does not contain every program point. The slice sets of P_2' and P_2'' are the same: $\texttt{SL}(6) = \{1, 2, 4, 5, 6, 11, 12, 14, 15\}$. Section 5.2 explains how to compute slice sets and how they can be used to obtain program slices.

Finally, we define a slice as being more *precise* than another if its slice set contains fewer elements. This notion of precision, though coarse, is sufficient for general comparisons between slices. Our proofs do not involve precision, only correctness. Nevertheless, considerations about precision are important when experimentally evaluating the result of the slicing, and they justify the usage of a CFG-preserving slice: both P_2' and P_2'' in Figure 5.1 share the same slice set, which implies they are equally precise. Therefore, we can use the CFG-preserving version (P_2'') without loss of precision.

Considerations about CFG-Preserving Slices

Program slicing is a transformation whose purpose is to *simplify* programs. The standard way to create a slice is to remove statements which are not in the slice set (those which are "useless" from the point of view of the slice). For instance, slices P_1' and P_2' in Figure 5.1 illustrate this approach: statements 3, 7, 8, 9, etc. have been completely removed from both slices. In this approach, the program slice has fewer statements than the original program, and its CFG may be significantly different (loops may have disappeared, `if` branches may have been merged, etc.). This is bothersome for our correctness proof, since these changes require reasoning about graphs and CFG paths.

Program slices can also be obtained via an equivalent transformation that preserves program

points, replacing statements not in the slice set with no-ops, except for conditions, which are replaced with *constant predicates*. A constant predicate is a conditional branch whose condition does not depend on any program variable: its result is statically known, and always evaluates to the same value. For instance, in the C language, the condition `if (1)` is a constant predicate.

A constant predicate is semantically equivalent to an unconditional jump. For instance, the constant predicate `if (1) goto A; else goto B` is semantically equivalent to `goto A`, and the constant predicate `if (0) goto A; else goto B` is semantically equivalent to `goto B`. The former is called an *always-true* constant predicate, while the latter is an *always-false* predicate.

We use constant predicates to perform *CFG-preserving transformations*, that is, to replace statements in a function without changing its control flow structure. Note that the control flow structure only changes if statements are replaced by other statements having a different number of successors. For instance, in the code `if (a < 10) b; else c;`, if we know `a` is in the interval $[10, 20]$, we can bypass the `if` and replace the entire code with `c;` without changing its semantics. But doing so would also change the control flow structure, since we have removed two nodes (the `if` condition and the `b;` statement). Instead, we may change the original code to `if (0) b; else c;`, which is semantically equivalent but does not change the control flow structure. This transformation is simpler to prove correct and does not affect the end result (e.g. the loop bound estimation, in our application of program slicing). In Figure 5.1, the instruction associated to program point 8 in P_2'' is a constant predicate. Note that this program slice is also a simplification of the original program, not because it has fewer statements, but because many of its statements actually do nothing.

In terms of efficiency, CFG-preserving slices are equivalent to standard slices with respect to the value analysis that is performed later. For the value analysis, constant predicates are equivalent to direct jumps. Since most of the cost of a value analysis is due to the presence of loops, the fact that these are short-circuited by constant predicates allows the value analysis to ignore them. In some experiments performed with a non-CFG-preserving slicer, there was no noticeable difference in terms of analysis time.

As a final note, we highlight the fact that a program slice obtained via a CFG-preserving transformation can always be converted into the slice produced by the former technique, by removing no-ops and short-circuiting constant predicates. This transformation is quite straightforward and not detailed here, since we do not make use of it.

5.2 Computing a Slice

In this section, we present all steps we performed to compute our program slices, to illustrate the concepts and difficulties involved in this process. These computations have been performed by non-verified code (in OCaml) and then validated *a posteriori*. This avoids formally specifying and proving each of the involved algorithms. Besides serving as indication on how to effectively compute a slice, this section justifies the use of validation to minimize proof effort.

The most common formulation of program slicing is based on the *program dependence graph* (PDG)[30], which represents the program as a directed graph where each CFG program point is a vertex and there is an arc between two vertices if the destination *depends* on the source. The notion of dependency employed here is the same as used in compiler literature. We consider two kinds of dependencies: *control dependencies* and *data dependencies*. The former are related to control flow, while the latter are related to variable values.

Figure 5.2 presents the architecture of the program slicer that will be described in this section. It is divided in two main parts, a *slice set calculator* and a *slice builder*. This separation is useful for the verified program slicer which will be presented in Section 5.3.

An advantage of using a PDG-based slicer is that the computation of the slice set is decomposed in two parts: first, the construction of the PDG, consisting of all dependencies between program points; then, a simple traversal of the PDG to obtain the slice set for a given criterion. Slice sets for different criteria in the same program can be obtained much faster, since the PDG does not need to be recalculated.

Example PDG Throughout this section, we use Figure 5.3 to illustrate each kind of dependency on the example program P_D. This program consists of a simple loop containing a conditional

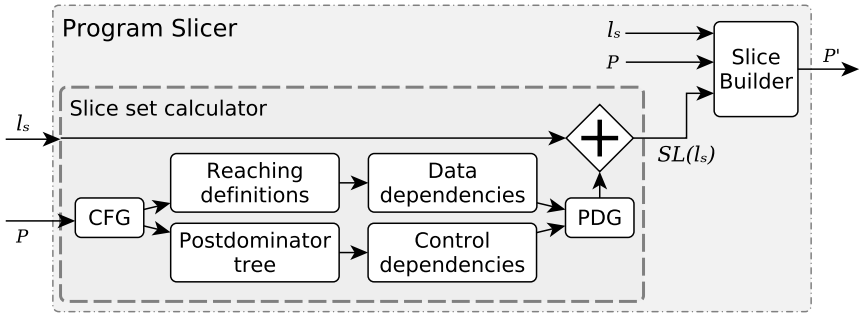


Figure 5.2: Architecture of a PDG-based program slicer. Each step used in the computation of the program slice is presented separately.

branch and some assignments. Its result is mostly nonsensical, but could be interpreted as the assignment of some values from variable `a` to variable `b`.

Figure 5.3 depicts the program's control flow graph (CFG), its *control dependence graph* (CDG), *data dependence graph* (DDG), then finally its program dependence graph (PDG). The last column highlights all nodes backward reachable from program point 3, which correspond to the slice set when the criterion is 3. Each of these graphs (except for the CFG) is detailed in the subsection corresponding to the dependencies it represents: control dependencies for the CDG, data dependencies for the DDG and the union of both for the PDG.

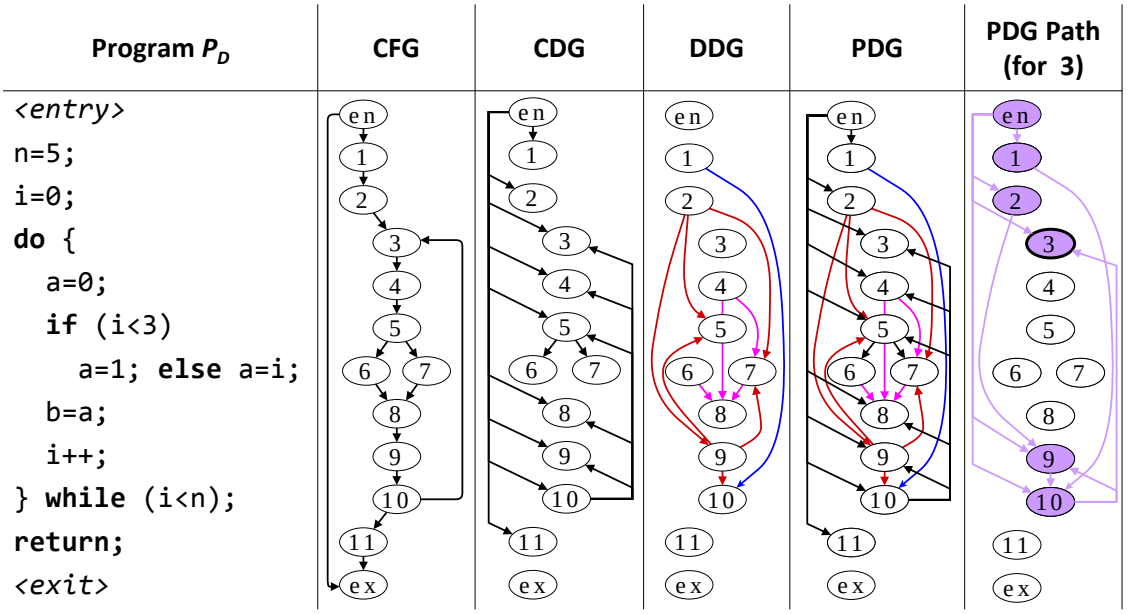


Figure 5.3: Example program P_D with its corresponding control flow graph (CFG), control dependencies (CDG), data dependencies (DDG), program dependence graph (PDG), and the result of the backwards traversal of the PDG, starting at the loop header at 3.

5.2.1 Control Dependencies

Control dependencies are related to the control flow between statements. Informally, a control dependency arises between two program points l and l' when l has at least two successors, one of which allows l' to be executed but not the other. In other words, l can "choose" whether l' will be executed. For structured programs, these dependencies can be computed syntactically: a loop condition controls the statements in the loop body, and an `if` condition controls the statements in its branches (the CFGs in Figure 5.4 illustrate both cases). Their computation requires only a

(a) (b)

Figure 5.4: Example CFGs to illustrate postdominance and control dependencies. In both CFGs, vertex **a** controls vertices **b** and **c**, but not **d**. The first CFG represents an `if` statement, while the second one represents a `while` loop.

top-down traversal of the syntax tree.

However, to deal with generalized CFGs containing unstructured control flow, the definition of control dependency requires the notion of *postdominance*.

Definition 1 (Postdominance)**.** *Let l and l' be two program points. l' postdominates l if, for every path ls from l to l_{exit}, $l' \in ls$.*

In Figure 5.4, l_{exit} is **d** in both CFGs. In Figure 5.4a, **d** postdominates every vertex (including itself). Neither **b** nor **c** postdominate **a**, since there is always a path to **d** that does not include one of these vertices. In Figure 5.4b, **a** postdominates **b** and **c**, since their only path to **d** includes **a**.

Postdominance defines a partial order on CFG vertices. We define *strict postdominance* as the irreflexive version of postdominance, that is, when l postdominates l' and $l \neq l'$. These definitions allow us to define control dependencies precisely.

Definition 2 (Control dependency)**.** *Let l and l' be two program points. l controls l' if: (1) $l \neq l'$, (2) l' does not strictly postdominate l, and (3) there exists a path ls from l to l' ($l \xrightarrow{ls} l'$) such that, for all $l'' \in ls \setminus \{l\}$, l'' is postdominated by l'.*

As an example, in Figure 5.4b, let us consider $l = \mathbf{a}$ and $l' = \mathbf{c}$, satisfying condition (1). **c** does not postdominate **a**, since the path [**a**] from **a** to the exit **d** does not include **c**. Since **c** does not postdominate **a**, it is also true that **c** does not *strictly* postdominate **a** (strict postdominance is a subset of postdominance), verifying condition (2) for control dependency.

We now verify condition (3) for control dependency. There is a path $ls = [\mathbf{a}, \mathbf{b}]$ from **a** to **c**. $ls \setminus \{\mathbf{a}\} = \{\mathbf{b}\}$, so we only need to verify the second condition for vertex **b**. Since all paths from **b** to **d** necessarily include **a** (**d**'s only predecessor), we conclude that **b** is postdominated by **a**. This verifies the last condition for control dependency, therefore **a** controls **c**.

To compute control dependencies, it is customary to consider an *augmented CFG*, which is equivalent to our normalized CFG (with unique entry and exit vertices, as described in Section 2.2.4) with an extra arc from l_{entry} to l_{exit}[2]. This arc prevents the need for special cases concerning the entry point and does not change the semantics of the program.

We describe here the control dependence algorithm used in [53]. The overall structure of the algorithm is the following:

postdominators → immediate postdominators (postdominator tree) → control dependencies

A more efficient, but more complex algorithm can be implemented using Lengauer and Tarjan's algorithm [43] for computing the postdominator tree. It replaces the first two steps in the above scheme.

We use the same program from Figure 5.3 to illustrate the construction of the control dependency graph. Figure 5.5 shows its CFG, its postdominator tree and the final CDG. Some CFG edges are colored red because they are special for the CDG computation, as we will see.

[2]Without this CFG arc, the control dependency graph would have no control arcs connecting l_{entry} to any vertex.

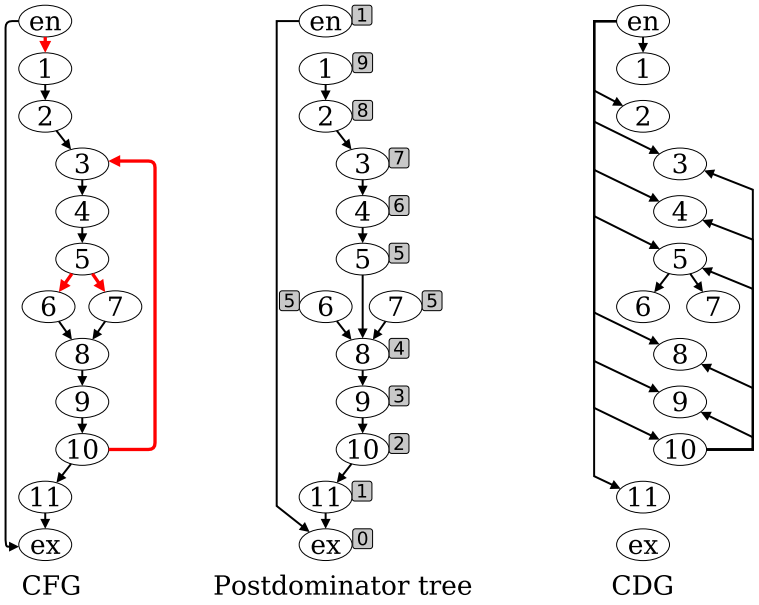


Figure 5.5: CFG of program P_D from Figure 5.3, its postdominator tree (with the cardinality of the postdominator set indicated next to each vertex) and its CDG.

Computing Postdominators Postdominators can be computed using a simple data-flow algorithm. Let *pdom(l)* represent the set of postdominators for program point l. We proceed as follows:

- starting from the program's CFG, we reverse the direction of each arc, obtaining the *reverse flow graph*[3];
- we define *pdom* via a set of data-flow equations whose greatest solution corresponds to the set postdominators of each program point:

$$\begin{cases} pdom(l_{\text{exit}}) = \{l_{\text{exit}}\} \\ pdom(l) = \{l\} \cup \left(\bigcap_{l' \in \text{preds}(l)} pdom(l') \right) \end{cases}$$

These data-flow equations can be computed using any data-flow solver. For instance, CompCert provides a module that can be instantiated with a set of *data-flow nodes* and a transfer function, and it generates an iterator that produces the desired solution. We instantiated it with CFG vertices as nodes and a trivial transfer function, corresponding to the *pdom* equations.

In Figure 5.5, the postdominator set of each vertex corresponds to all reachable vertices in the postdominator tree. The gray rectangles indicate the cardinality of these sets. For instance, $pdom(6) = \{8, 9, 10, 11, l_{\text{exit}}\}$.

Immediate Postdominators (Postdominator Tree) After obtaining the set of all postdominators, we need to reduce it to the set of *immediate* postdominators, named *ipdom*. In terms of relations, *ipdom* is the *transitive reduction* [1] of *pdom*, the smallest relation that preserves the transitive closure. This amounts to removing all arcs (a, c) where $b \in pdom(a)$ and $c \in pdom(b)$. For the algorithm, a more efficient implementation consists in comparing postdominator set sizes. By using the following two facts:

- $b \in pdom(a) \ \wedge \ c \in pdom(a) \Rightarrow c \in pdom(b) \ \vee \ b \in pdom(c)$ (all postdominators of a node are related by postdominance themselves);

[3]We are actually going to compute the set of *dominators* on the reverse CFG, which is equivalent to the set of postdominators in the original CFG.

- $c \in pdom(b) \Rightarrow |pdom(b)| \geq |pdom(c)|$ (a vertex always has more postdominators than any of its own postdominators);

we obtain that the *immediate* postdominator of l is the one having maximum cardinality among its own postdominators.

Each vertex (except l_{exit}) has a single immediate postdominator. This produces a *tree*, such as the one in Figure 5.5, which is a compact representation of the postdominance relation.

Computing Control Dependencies The CFG and the postdominance tree are both used to compute the control dependency graph. An efficient (though unintuitive) algorithm to do it is the following: we iterate through each edge (a, b) in the CFG, and check if b does not postdominate a. Such edges are indicated in red in Figure 5.5. For instance, let us consider edge $(10, 3)$ in the CFG. We then look for the *lowest common ancestor* of a and b in the postdominator tree. Let us call it *lca*. For vertices 10 and 3, the lowest common ancestor of both is 10 itself[4]. We now climb the tree, from b up to and excluding *lca*, and for each vertex x in the path, we claim that a controls x, and add an edge (a, x) in the CDG. In our example, this results in adding arcs 10→9, 10→8, 10→5, 10→4 and 10→3 in the CDG. By repeating the process with edges $(l_{\text{entry}}, 1)$, $(5, 6)$ and $(5, 7)$, we obtain all arcs in the final CDG.

Though unintuitive, this algorithm does follow the definition of control dependencies and computes correct results. Its efficiency is due to the fact that we can ignore all edges (a, b) where a is postdominated by b, and this is the most common case for CFG edges.

Another algorithm to efficiently compute control dependencies, given the tree of immediate postdominators, is described in [20]. Combining it with Lengauer and Tarjan's algorithm to obtain the immediate postdominators, this constitutes an alternative implementation to compute control dependencies.

5.2.2 Data Dependencies

Data dependencies are related to the flow of values between program points. Their formalization is based on the *use* and *def* sets defined for each program statement: if x is defined in l, used in l' and there exists a CFG path from l to l' such that x is not redefined along this path, then there is a data dependency from l to l'.

Definition 3 (Data dependency)**.** *Let l and l' be two program points.* l influences l' *(reciprocally, l' is data dependent on l) if there exists a variable x such that $x \in \texttt{def}(l) \cup \texttt{use}(l')$ and there exists a path ls from l to l' ($l \xrightarrow{ls} l'$) such that, for all $l'' \in ls \setminus \{l\}$, $x \notin \texttt{def}(l'')$.*

Computing data dependencies The computation of data dependencies is performed using a *reaching definitions* analysis [56]. This analysis requires a data-flow solver (such as the one used to compute postdominators for control dependencies) and the equations it uses are very similar to the definition of *def/use* sets in the language. It computes, for each program point l', a set of pairs (x, l) indicating that there is a definition of x at program point l and this definition is reachable at l'.

At each program point, the data-flow solver simply propagates all pairs (x, l) at the current program point for variables that have not been redefined. For instance, let us suppose that at program point 8 there is a statement $\texttt{assign}(x, e, 9)$ (i.e. the equivalent of the following C code: `x = e; goto 9`), and that the data-flow solver has the set of pairs $RD(8) = \{(x, 6), (x, 7), (y, 4)\}$ before entering 8. The solver will propagate to program point 9 the following information: $RD(9) = \{(y, 4), (x, 8)\}$. The previous definitions of x at 6 and 7 are not reachable at 9.

Concerning our implementation of this analysis, two aspects are worth mentioning:

- if there is a considerable number of variables (i.e. thousands), the analysis can be costly in terms of memory and time, for instance due to the creation of temporaries. Since a temporary variable is defined a few times and then never redefined, it may cause a quadratic increase in the number of reachable pairs. To prevent this from happening, before the reaching definitions

[4] It is not always the case that one of a or b is the lowest common ancestor. For instance, if we consider edge $(5, 6)$, the lowest common ancestor is 8.

analysis we compute a *liveness analysis*, which is less expensive, and then use its result to avoid propagating information about dead variables (i.e. variables which are no longer used). This maintains the size of reachable sets roughly linear in the number of program points;

- the *Mem* variable representing the memory needs special treatment: when there is a `store` instruction, it only redefines *part* of the memory (this is called a *weak update*). Therefore, it is incorrect to assume that any previous definitions of *Mem* are unreachable. In this case, we deviate from the *def* function and keep the previous definitions of *Mem* reachable. Although less precise, this is needed for a correct analysis. In Section 5.5, we mention some solutions to improve this result.

The second step of the computation of data dependencies is the conversion of *reaching definitions* into a Data Dependency Graph (DDG), where each arc (l, l') indicates that *there exists a variable, defined in l, such that its definition is reachable at l', where it is used.* Note that here we do not need to know precisely *which* variable is reachable.

To compute this information, our algorithm iterates through each program point l' and through each variable $x \in \texttt{use}(l')$, and computes the union of all program points $\{l | (x, l) \in RD(l')\}$. Then, for all such program points l, it adds an arc (l, l') to the DDG. This is depicted in the fourth column of Figure 5.3 for program P_D.

5.2.3 Union of Dependencies

A program slice contains both control and data dependencies related to the slicing criterion. The fifth column in Figure 5.3 contains the Program Dependence Graph (PDG) related to program P_D. This graph is the union of the previous dependency graphs. To perform this union, since both graphs have the same sets of vertices, we simply have to perform the union of arcs: there is an arc (l, l') in the PDG if there is such an arc (l, l') either in the CDG or in the DDG.

Obtaining the Slice Set To consider all dependencies from the beginning of the program execution until the slicing criterion, it suffices to perform a backward traversal of the PDG, starting at the slicing criterion and advancing through dependency arcs: if the PDG contains an arc (l, l') and the traversal has reached l', then it can reach l as well. The set of vertices reached this way constitutes the slice set.

Note that (as depicted in Figure 5.2) the dependency graph is independent of the slicing criterion. Therefore, to compute other slice sets for the same program, the only necessary recomputation is the backward traversal. After obtaining the slice set, we use it to compute the program slice itself.

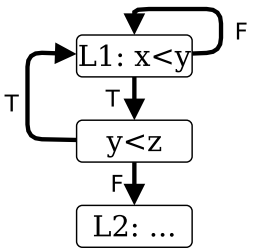
5.2.4 From Slice Set to Program Slice

The function that, given a slice set and the original program, produces an executable program slice is called *slice builder*. It is much simpler than the slice set calculator, but it is not always a trivial function.

In a structured language like C without `goto`s, a slice builder is very simple: it only needs to iterate over the statements in the slice set, removing from P any statement not in $\texttt{SL}(l_s)$. A CFG-preserving slicer for this language might just replace any sliced statement with either *skip*, or an *always-false* constant predicate. This works because in a structured language the slice builder can easily remove entire `if` blocks and loops, or simply replace their conditions with `false`.

A slice builder for an unstructured language such as C with `goto`s, or CompCert Cfg, is more complex, however. The main difficulty comes from the slicing of conditional statements. Arbitrary control flow means that we never know if the branch to take is the `true` one or the `false` one. For instance, consider the following C program:

```
L1:
  if (x < y)
    if (y < z) goto L1;
    else goto L2;
  else
    goto L1;
L3:
  exit;
L2:
  print(1);
  ...;
```



This program contains a loop that starts at `L1` and has `L2` as its only exit. L3 is unreachable in this program. L2 is reached when the first condition ($x < y$) is true and the second one ($y < z$) is false.

Let us slice this program with respect to L2. L2 has no dependencies from any statement in the loop, so the entire loop can be sliced away. This means the slice builder may replace both conditional statements with constant predicates. $x < y$ must be replaced with an *always-true* constant predicate, otherwise it loops in L1. $y < z$ must be replaced with an *always-false* predicate, otherwise it also loops in L1. Note that we cannot simply remove both `if`s. In this case, program point L3 would be reached, instead of L2.

We want a systematic way to efficiently decide the branch for each constant predicate. For now, we suppose that there is an *oracle* which gives us this information. We call this oracle `branch_oracle`, a function that, given both successors l_{true} and l_{false} of a branching instruction, returns which branch must be chosen. This oracle is integrated into the pseudocode of the `slice_builder` function presented in Figure 5.6. This function, whose inputs are the program `P` and the slice set `SL_ls`, produces the executable program slice.

The slice builder algorithm is not as straightforward as in structured code, but it is still relatively simple: for each program point l, it checks if l belongs to the slice set. If so, it preserves the statement associated to l (line 4). Otherwise, it replaces this statement with:

- a no-op, if the statement is not a condition (line 7);
- a constant predicate otherwise (lines 9-11). The `constant_predicate` function, using the result of the branch oracle, assigns either an *always-true* predicate or an *always-false* predicate to the sliced condition.

Note that this CFG-preserving slice builder can be transformed into a standard slice builder by directly connecting the predecessors of the statement at l to the branch returned by the branch oracle.

```
Definition slice_builder (P : program) (SL_ls :  list vertex) :=
1   foreach program point l in P:
2     let s := P.code[l] in
3     if l ∈ SL_ls:
4       P'.code[l] := s
5     else:
6       if s is an unconditional statement:
7         P'.code[l] := nop
8       else:  (* inst is a condition *)
9         let ([l_true, l_false] := successors(l)) in
10        let (b := branch_oracle(l_true, l_false) in
11        P'.code[l] := constant_predicate(b, l_true, l_false)
```

Figure 5.6: Pseudocode of a slice builder algorithm, which uses a slice set to produce a program slice.

Computation Cost The computation cost of the slice builder is negligible when compared to the computation of the slice set. The slice builder only requires local computations on each program point. We will see that this is also true for the `branch_oracle` function, whose computation cost is unknown for now.

5.3 Correctness of Program Slicing

After computing the slice set, we must ensure it is correct. This section presents a theorem of correctness for program slices, and then details the technique used to prove our program slicer is correct. Although the theorem presented in this section is not the one actually present in the formal Coq development, the main reason we present it here is to illustrate a more general notion of correctness for program slicing which does not depend on our specific use case (preservation of execution counters). In the interest of a possible future usage of program slicing as part of another formal development, this notion of correctness constitutes the most general preservation property about program slicing.

The overall proof idea of slicing correctness, based on the works of Ranganath et al. [61] (pen-and-paper proofs) and Wasserrab [67] (Isabelle/HOL formalization[5]), has been formalized in Coq and extended to accommodate a constructive approach (to enable automatic code generation) based on *a posteriori* validation. In particular, the architecture presented in Section 5.3.1 has been developed in this thesis, with an axiomatization suitable for efficient validation. The integration of the correctness results with the loop bound estimation presented in Section 5.4 are also novel work developed in this thesis.

The slice behaves exactly as the original program with respect to the slicing criterion l_s. We express this behavior as the *preservation of the observable trace for relevant variables*. By *trace*, we mean the sequence of semantic states traversed during program execution. By (1) *observable*, we mean that we only consider vertices *in the slice set*, and (2) *for relevant variables* means that we consider a specific subset of the program variables, ignoring the values of the others: for each program point l, we define a set $\texttt{RV}(l)$ of *relevant variables at* l which contains all variables possibly influencing the slicing criterion, due either to control or to data dependencies. We will later detail how the $\texttt{RV}(l)$ sets are computed. Note that program slicing does *not* preserve the observable *events* trace (defined in Section 2.2.2). It is entirely possible (and often intended) that program slicing will eliminate observable events, such as volatile memory loads. This fact justifies the need for a more convoluted definition of correctness, depending on the notion of relevant variables.

Let us consider the program P_T in Figure 5.7, with its program slice P'_T with respect to vertex 4. The execution trace tr of this program is the following[6], using the notation l:*(i,j,k)* to indicate that the current program point is l, and i, j, k are the respective values of these variables:

$$tr = \{1{:}(?,?,?), 2{:}(?,?,?), 3{:}(?,?,2), 4{:}(0,?,2), 5{:}(0,?,2), 6{:}(0,?,2), 8{:}(0,1,2), 9{:}(0,1,3), 10{:}(1,1,3), \\ \ldots, 10{:}(3,2,6), 11{:}(3,2,6), 12{:}(3,2,9)\}$$

Let us now apply each of the restrictions concerning the preservation guaranteed by the slicing. First, we apply (1), that is, we filter the trace based on the slice set. The slice set of P'_T is $\{1, 3, 4, 9, 10\}$, therefore the filtered trace $(tr)_1$ is:

$$(tr)_1 = \{1{:}(?,?,?), 3{:}(?,?,?), 4{:}(0,?,?), 9{:}(0,?,?), 10{:}(1,?,?), 9{:}(1,?,?), 10{:}(2,?,?), 9{:}(2,?,?), 10{:}(3,?,?)\}$$

Note that the values of some variables (namely, j and k) have changed with respect to the original trace: j and k in particular are no longer initialized. This is not an issue, however.

The application of restriction (2) means reducing each semantic state to $l{:}i$, since i is the only relevant variable in the slice set. Trace $((tr)_1)_2$ becomes:

$$((tr)_1)_2 = \{1{:}?, 3{:}?, 4{:}0, 9{:}0, 10{:}1, 9{:}1, 10{:}2, 9{:}2, 10{:}3\}$$

This is the trace *preserved* by the slicing, which means it is the same for both P_T and P'_T. Let the trace of P'_T be *tr'*. Then we have $((tr)_1)_2 = ((tr')_1)_2$. Note that $tr \neq tr'$ and $(tr)_1 \neq (tr')_1$.

[5] This formalization is non-executable (due to the use of e.g. existential quantification) and cannot be used to automatically generate executable code.

[6] We abstract information related to the memory state.

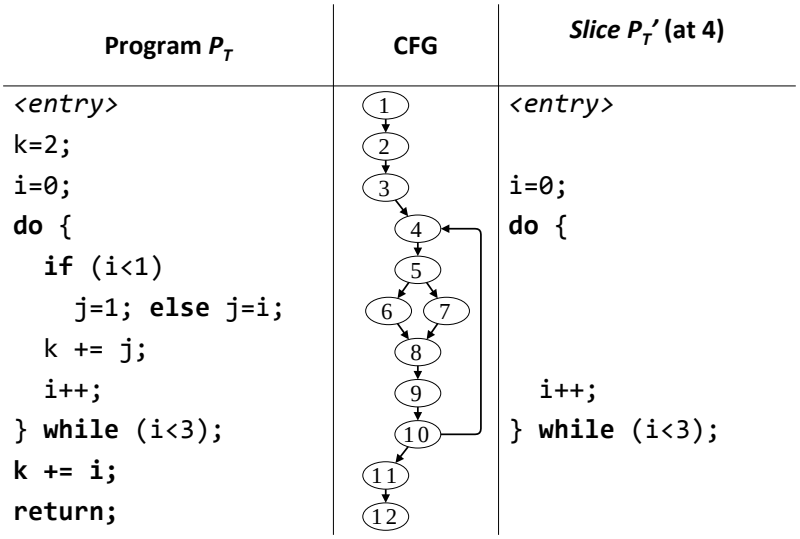


Figure 5.7: Example program P_T, its control flow graph and its slice P'_T with respect to vertex 4.

Formally, we can state the soundness of program slicing with respect to relevant variables as in Theorem 4 below. Note that this theorem has a quite complex formulation. For this reason, we prefer to define application-specific correctness theorems, according to each usage of program slicing. These theorems use the results of Theorem 4, but they can be expressed more clearly in their applications.

Theorem 4 (Soundness of Program Slicing with respect to Relevant Variables)**.** *Let P be a program and l_s a program point of P.*
Let P' be the slice of P with respect to the slicing criterion l_s.
For any reachable state $\sigma \in \texttt{reach}(P)$ such that its program point is in the slice set ($\sigma.l \in \texttt{SL}(l_s)$), the same program point can be reached in P', with the same values for all relevant variables:

$$\forall \sigma \in \texttt{reach}(P), \sigma.l \in \texttt{SL}(l_s) \Rightarrow$$
$$\exists \sigma' \in \texttt{reach}(P') \ \wedge \ \sigma'.l = \sigma.l \ \wedge \ \forall x \in \texttt{RV}(\sigma.l), \sigma.E(x)(\sigma.l) = \sigma'.E(x)(\sigma'.l)$$

In the formal Coq development, this property is not stated directly, because it is implicit in the simulation used to prove a different correctness criterion. This other criterion is related to the application of program slicing in the loop bound analysis and can be stated as *the preservation of execution counters for the slicing criterion*. This theorem and its correctness are presented in Section 5.4.

5.3.1 *A Posteriori* Validation of Program Slicing

The concept of *a posteriori* validation has been introduced in Section 2.1.2. The computation of a slice set, as described in Section 5.2, is a good candidate for validation: there are efficient, heuristic-based techniques to compute it; the proofs of some of these techniques (e.g. postdominator trees) are complex; and the necessary properties can be validated efficiently, as we will see. For these reasons, instead of a direct proof of Theorem 4, we opted for the application of *a posteriori* validation for the slice set calculator, and then the use of a proved component to conclude the slicing.

Our validation approach consists in the following steps, some of which are depicted in Figure 5.8:

1. Defining a set of properties necessary for the correctness proof of program slicing. For instance, one such property is "the slice set must contain the slicing criterion". These properties are stated in terms of the slice set and in terms of additional information that must be computed for the proof (the slice set itself does not contain enough information for the proof). We call this information `Extra` and will detail it later.

2. Implementing an untrusted *slice set calculator* that, given a program P and a slicing criterion l_s, yields a slice set $\texttt{SL}(l_s)$, plus the `Extra` information associated to it.

3. Specifying and implementing (in Coq) a *checker* that, given $\texttt{SL}(l_s)$ and `Extra`, performs some tests on both and returns `true` if they pass the tests.

4. Proving (in Coq) the correctness of the checker, which ensures that if the checker outputs `true`, then the slice set respects the properties defined in item 1.

5. Implementing (in Coq) a *slice builder*: a proved function that, given a program, a slicing criterion, a checked slice set and `Extra`, computes a program slice.

6. Using the correctness of the checker to prove the correctness of the builder: any slice set that passes step 3 results in a correct slice.

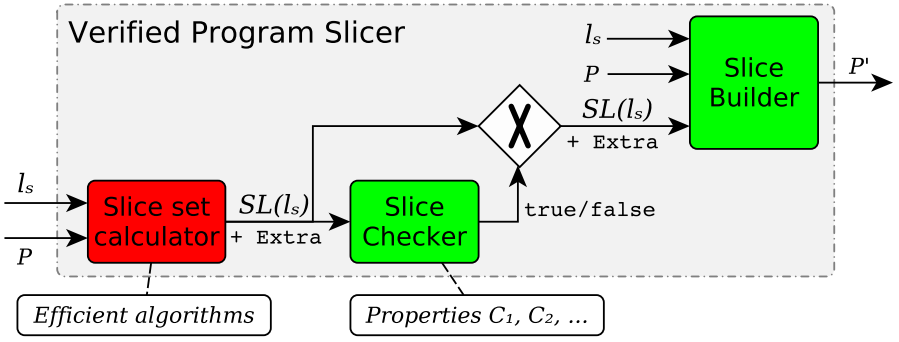


Figure 5.8: Architecture of the validated program slicing. Red blocks are untrusted code, green ones are proved code. From the inputs P (the program to be sliced) and l_s (the slicing criterion), we compute the slice set $\texttt{SL}(l_s)$, check it, and then build the slice P'.

In this section, we detail each of these steps. We present the properties used in the correctness proof of program slicing and also some implementation details of the checker, such as completeness and efficiency. They are important for the experimental evaluation of this work. The overall architecture of this program slicer is presented in Figure 5.8.

Axiomatization of Properties on Slices

The properties we need to validate depend on the correctness proof technique of our choice. First we describe this technique, then we formalize some concepts used in this proof. Afterwards, we present and detail the axiomatization, that is, the list of properties.

We use a *weak simulation* based on paper-and-pencil proofs of program slicing [61]. These proofs use mainly two kinds of information related to the slice set: relevant variables and *next observable vertices* (formally defined below). Relevant variables are related to data dependencies, while next observable vertices are related to control dependencies. Note that the proof does not consider *how* to obtain the slice set; it supposes such a slice set exists, and proceeds on how to prove it correct.

Definition 4 (Next Observable Vertices)**.** *For a given program P, we associate to some program points l of P a next observable vertex $\texttt{NObs}(l)$. This is the closest vertex (in terms of CFG oriented distances) to l that belongs to the slice set $\texttt{SL}(l_s)$. In particular, if $l \in \texttt{SL}(l_s)$, then $\texttt{NObs}(l) = l$.*

Next observable vertices are useful for an efficient construction of the CFG-preserving slice, besides being necessary for the correctness proof. They inform the slice builder about which successor to choose next, when there are multiple choices (e.g. in a conditional branch). Some examples of next observable vertices are presented in Figure 5.10. Note that `NObs` defines a *partial* function from vertices to vertices. In particular, for a vertex that cannot reach any vertex in the slice set, there is no next observable associated to it. Such a vertex is considered *after the slice*.

(C_1) $l_s \in \texttt{SL}(l_s)$

(C_2) If $n \in \texttt{SL}(l_s)$, then $\texttt{use}(n) \subseteq \texttt{RV}(n)$

(C_3) If $n \mapsto s$, then $\begin{cases} \text{if } n \notin \texttt{SL}(l_s), \text{ then} & \texttt{RV}(s) & \subseteq \texttt{RV}(n) \\ \text{otherwise} & \texttt{RV}(s) \backslash \texttt{def}^*(\texttt{n}) & \subseteq \texttt{RV}(n) \end{cases}$

(C_4) If $\texttt{def}(n) \cap \texttt{RV}(n) \neq \emptyset$, then $n \in \texttt{SL}(l_s)$

(C_5) $n \in \texttt{SL}(l_s) \iff \texttt{NObs}(n) = n$

(C_6) If $n \notin \texttt{SL}(l_s) \wedge \texttt{NObs}(n) = o$, then $\forall s, n \mapsto s \Rightarrow \texttt{NObs}(s) = o$

(C_7) If $n \notin \texttt{dom}(\texttt{NObs}) \wedge n \mapsto s$, then $s \notin \texttt{dom}(\texttt{NObs})$

(C_8) $n \in \texttt{SL}(l_s) \iff \texttt{DObs}(n) = 0$

(C_9) If $n \in \texttt{dom}(\texttt{NObs})$, then $\texttt{DObs}(n) \geq 0$

(C_{10}) If $n \notin \texttt{SL}(l_s) \wedge \texttt{DObs}(n) = d$, then $\exists s, n \mapsto s \wedge \texttt{DObs}(s) < d$

Figure 5.9: Axiomatization of slices, relevant variables, next observable vertices and distances.

When execution reaches a vertex after the slice, it means that the program will never again reach any vertex in the slice set.

The definition of next observable vertices involves *CFG distances*. The CFG distance between two program points is the length of the shortest directed path along CFG arcs. For instance, in Figure 5.7, the CFG distance between program points 3 and 5 is 2 (the shortest path has two edges, 3-4 and 4-5), but between 5 and 3 the CFG distance is 5 (the shortest path goes all the way from 5 to 10, then takes the edge 10-3).

The `NObs` function returns a program point, but not the distance to it. The *next observable distance* function `DObs` captures this information. `DObs` is a partial function from vertices to distances. Both `NObs` and `DObs` are used in the construction and proof of correctness of our program slicer. The correctness proof uses these three sets, `RV`, `NObs` and `DObs`, so our slice set calculator must construct them for the validation process. The `Extra` information in Figure 5.8 is in fact constituted of these three sets. The axiomatization defines which properties must be checked on them.

Axiomatization Our axiomatization, which also serves as the specification of our checker, is defined as a set of constraints on the sets $\texttt{SL}(l_s)$, `RV`, `NObs` and `DObs`, as well as on the *use* and *def* sets (described in Section 2.2.4). The main constraints are presented in Figure 5.9. l denotes a program point, l_s the slicing criterion and $n \mapsto s$ denotes that s is a successor of n in the CFG.

Property (C_1) states the slicing criterion is in the slice set. Properties (C_2) to (C_4) show that $\texttt{RV}(l)$ and $\texttt{SL}(l_s)$ are mutually dependent sets: $\texttt{RV}(l)$ contains the variables that are defined in $\texttt{SL}(l_s)$ and whose value may affect the execution of the statements in $\texttt{SL}(l_s)$, while $\texttt{SL}(l_s)$ contains every statement assigning variables in $\texttt{RV}(l)$.

Property (C_2) states that any variable that is used in the slice set must be a relevant variable. Property (C_3) expresses the backward propagation from s to n. Vertices not in the slice inherit all relevant variables of their successors, while vertices in the slice only inherit those which they do not redefine themselves. The notation $\texttt{def}^*$ is defined as $\texttt{def}(n) \setminus \mathit{Mem}$. It means that the memory variable *Mem*, once relevant, remains relevant even if redefined. This is due to the fact that the memory is only *weakly updated*, that is, only part of it is redefined by the statement (this has been explained in the computation of data dependencies, in Section 5.2.2). Finally, (C_4) states that any vertex n defining relevant variables belongs to the slice set.

We can see properties (C_1) to (C_4) as a backward data-flow algorithm:

- the algorithm starts at the slicing criterion, applies (C_1) and then (C_2), which adds some elements to $\texttt{RV}(l_s)$;

- then, propagating (C_3) backwards, collects relevant variables for other program points;
- by applying (C_4) and (C_2), and alternating with the previous step, it keeps incrementing the relevant variable sets until a fixpoint satisfying all constraints is reached.

The properties that follow axiomatize next observable vertices and their distance. Property (C_5) states that any vertex in the slice set is its own next observable. Property (C_6) states that the observable vertex o of a vertex n that is not in the slice set is the same for all successors of n. This is an important and non-trivial property, a consequence of the fact that each program point has a unique next observable vertex. Property (C_7) is related to vertices having no next observable vertex: none of their successors has a next observable vertex either.

Properties (C_8) to (C_{10}) are related to the distance of next observable vertices. (C_8) is a trivial counterpart to (C_5): a vertex in the slice set has zero distance to itself. (C_9) ensures that any vertex having a next observable also has a finite and non-negative distance to its next observable. Finally, (C_{10}) states that any vertex n not in the slice set that has a defined distance must have at least one successor closer to its next observable.

Checker Algorithm

The checker is a short program that can be derived almost mechanically from the previous axiomatization. It consists in traversing the program's CFG, checking for each vertex and for its immediate successors if constraints (C_1) to (C_{10}) are respected in the given (untrusted) sets $\texttt{SL}(l_s)$, `RV`, `NObs` and `DObs`.

One notable aspect of the checker is its efficiency: the axiomatization of the sets to be validated only deals with *local* properties at each program point, which are then used to construct *path* properties such as "there always exists a path in the CFG from a vertex to its next observable". Since the local properties are stated in terms of vertices and their immediate successors, validation overhead is minimized, which is important for the performance of the extracted algorithm.

We also claim (without formally verifying) the completeness of our checker. As indicated in Ranganath et al. [61], the computation of a slice based on control and data dependencies entails the preservation of the sets of relevant variables. The other checked properties are related to the sets of next observable vertices, which are themselves computed from the control dependencies, therefore always valid for the program slices we compute. Experimental evaluation agrees with this claim: no correct slices have ever been rejected by our checker.

Building the Verified Program Slice In Figure 5.8, we see that after computing the untrusted slice set and checking its correctness via the slice checker, we obtain a trusted slice set and the `Extra` information, which has also been checked for correctness. The next step is the construction of the program slice itself, using the original program P.

We presented in Section 5.2.4 an unproved slice builder algorithm, mentioning the presence of a *branch oracle* without detailing its implementation. The slice builder is a function simple enough that it can be proved correct with minimal effort, as long as we have the necessary properties about the branch oracle. Namely, that *the branch the oracle chooses does not lead to the creation of infinite loops*. As presented in the example in Section 5.2.4, if the branch oracle leads to the creation of an infinite loop, the program slice will be incorrect. We present here how the oracle actually computes its choice, and why it prevents infinite loops.

The `branch_oracle` function uses the next observable distance `DObs` to choose the correct branch: the successor having the smallest distance to its next observable vertex is always chosen (in case of equality, we arbitrarily choose the `true` branch). Thus, `branch_oracle` can be defined simply as the return value of the comparison $\texttt{DObs}(l_{\texttt{true}}) \leq \texttt{DObs}(l_{\texttt{false}})$.

The reason this oracle works is related to the following fact *(f_l): when a loop condition is not in the slice set, its next observable vertex is not in the loop*[7]. This is a consequence of the fact that loop conditions *control* statements in the loop body. Since we are replacing a condition at a vertex l that has been sliced away (that is, $l \notin \texttt{SL}(l_s)$), we know that its next observable vertex is not in the loop.

[7] We do not prove this property, since this is not needed for the correctness of the slicing; the validation approach allows us to avoid having to prove properties such as this one.

The next observable distance, by definition, always decreases for at least one successor. By always picking the successor with smallest distance, we are certain that `DObs` will reach zero and we will eventually reach this next observable vertex. Since it is outside the loop (due to property f_l), we will reach the loop exit. Reaching its exit means we avoided creating an infinite loop, which is the property we needed the branch oracle to ensure. Figure 5.10 presents an example program P_S with its CFG and the computation of next observable distances for some conditional branches. It shows that in some cases the oracle chooses the `true` branch, while in others it selects the `false` branch. The `DExit` distances in the figure are not relevant here (they are presented in Section 5.4.1). Note that we also rely on determinism to ensure that the absence of infinite loops in the original program entails their absence in the program slice.

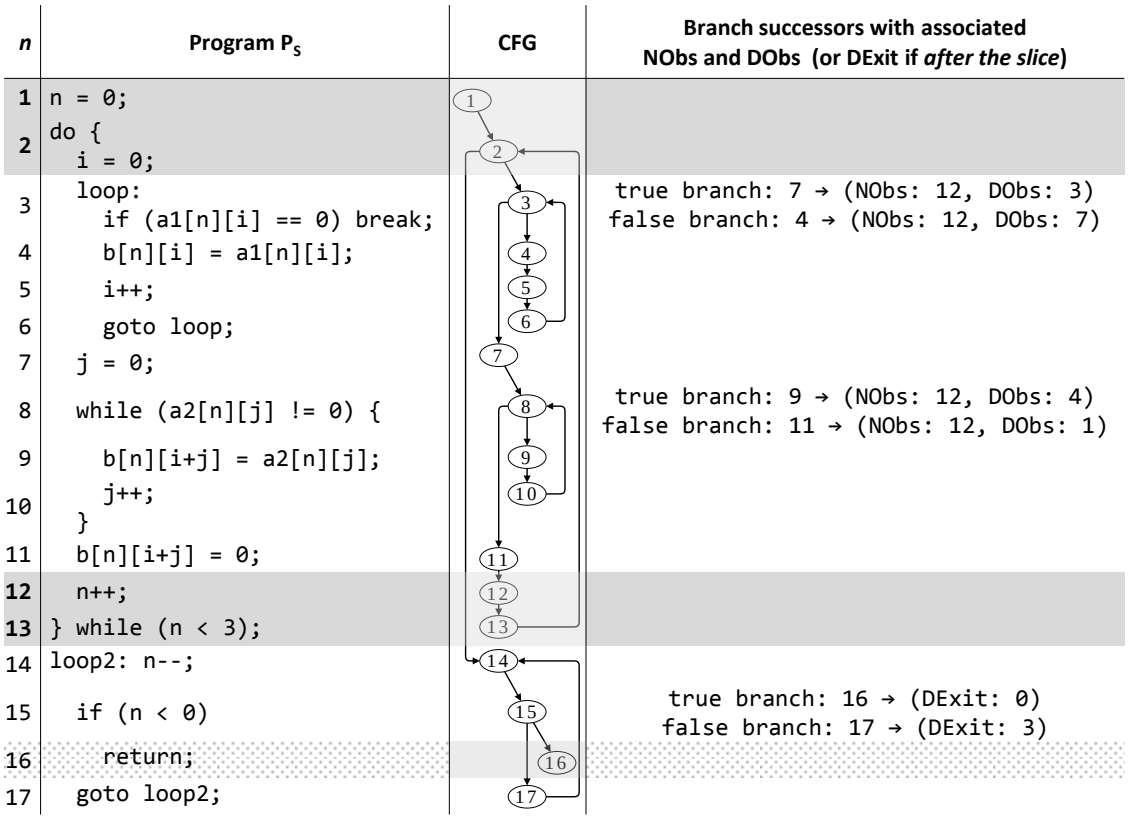


Figure 5.10: Choice of direction for constant predicates: at each branching point not in the slice set, we inspect both successors' next observable distances (`DObs`) and choose the one closest to its next observable vertex. For vertices *after the slice*, we choose the successor closest (via `DExit`) to the exit vertex.

5.3.2 Proof by Simulation

The slice checker axiomatization gives us several properties about the program slice, but we still need to apply them to obtain a proof of correctness for the slicing theorem.

Soundness of program slicing is classically proved by a *simulation* between the original program and its slice: by relating execution states in the program P to the corresponding states in the slice P', we can prove Theorem 4.

To account for the differences between the initial program and its slice, we define a matching relation between execution states σ and σ'. This relation is written $\sigma \sim \sigma'$ and its definition is presented in Figure 5.11. As is often the case, the main difficulty of the proof is to define the correct relation, since the proof technique does not help finding it. Correctness can then be proved by showing that, whenever we can perform an execution step in the original program, there is a corresponding situation in the program slice that allows it to execute a number of steps and to

preserve the matching relation $\sim$ between states in the programs. The formal statement of this simulation is presented in Lemma 4.

$$\frac{l \in SL(l_s) \quad E \simeq_{RV(l)} E'}{(l, E) \sim (l, E')} \ (R_1)$$

$$\frac{l \notin SL(l_s) \quad l' \notin SL(l_s) \quad \texttt{NObs}(l) = \texttt{NObs}(l') \quad E \simeq_{RV(l)} E'}{(l, E) \sim (l', E')} \ (R_2)$$

$$\frac{l \notin \texttt{dom}(\texttt{NObs}) \quad l' \notin \texttt{dom}(\texttt{NObs})}{(l, E) \sim (l', E')} \ (R_3)$$

Figure 5.11: Matching between execution states of a program and one of its slices (whose slice set is $SL(l_s)$).

To simplify Figure 5.11, execution states are considered as pairs (l, E), comprised of a program point and an environment; other state components are omitted. Given a program point l, we use $\simeq_{RV(l)}$ to denote the equivalence relation between two environments restricted to relevant variables at l.

The first rule (R_1) matches intuitively an execution state of the initial program with an execution state of the slice when the program point l is the same in both states and it belongs to the slice set $SL(l_s)$: both states match when the relevant variables have the same values in both environments E and E'.

The second rule (R_2) matches two states such that neither of their program points l and l' belong to the slice set $SL(l_s)$, but at least one of their successors belong to $SL(l_s)$. This(these) successor(s) are precisely identified using next observable vertices. Both states match when the next observable vertices at l and l' are the same and, as in the first rule, the relevant variables have the same values in both environments E and E'.

The final rule, (R_3), deals with vertices after the slice. It is a generalization of (R_2): at this point, $\texttt{RV}(l)$ is always empty and relevant variables can no longer be modified. Any two states after the slice are considered as matching.

These rules allow us to prove Lemma 4, which states that assuming the constraints of Figure 5.9, the sliced program executes in ways that simulate the execution of the corresponding initial program. Theorem 4 can be stated as a corollary of Lemma 4. We note in Lemma 4 an *execution step* in the program semantics as $\rightarrow$. Its reflexive transitive closure is written as $\rightarrow^*$.

Lemma 4 (Simulation of Program Slicing)**.** *Let P be a program and l_s a program point of P. Let the result of slicing P with respect to l_s using an untrusted slicer be $(P', \texttt{SL}(l_s), \texttt{RV}, \texttt{NObs}, \texttt{DObs})$. Assume $(\texttt{SL}(l_s), \texttt{RV}, \texttt{NObs}, \texttt{DObs})$ satisfy the constraints (C_1) to (C_{10}).*
$\forall \sigma_1, \sigma_2 \in \texttt{reach}(P), \sigma_1' \in \texttt{reach}(P')$, if $\sigma_1 \rightarrow \sigma_2$ and $\sigma_1 \sim \sigma_1'$,
there exists σ_2' such that $\sigma_1' \rightarrow^* \sigma_2'$ and $\sigma_2 \sim \sigma_2'$.

This concludes the proof of the correctness theorem of program slicing. However, as it happens quite often in verified program development, it is not sufficient to have the correctness theorem of a single component *per se*, since the integration of the analysis (in our case, program slicing) often requires additional properties for the proof of the entire development to succeed. This is the case in our loop bound analysis, which uses an *instrumented* semantics. This semantics requires additional properties about slices to be able to relate them to the execution counters of the loop bound analysis. We describe these requirements, the new correctness theorems, and the changes to the slicing framework on the next section.

5.4 Correctness of Program Slicing for a Loop Bound Analysis

Program slicing is a component of our verified loop bound analysis. As such, we state its correctness with respect to the loop analysis. The important information in this analysis is related to the preservation of execution counters of the slicing criterion (execution counters are part of the instrumented semantics presented in Section 4.4.1). The formal statement of this correctness property is Theorem 5 below.

Note that our semantic states here are those of the *Instrumented Cfg* language: they have two additional elements, the sets of local counters c_{loc} and global counters c_{glob}.

Theorem 5 (Soundness of Program Slicing with respect to Local Execution Bounds)**.**
Let P be a program and l_s a program point of P.
Let P' be the slice of P with respect to the slicing criterion l_s.
If M is a bound of every reachable local counter at l_s in P', that is,

$$\forall\ \sigma \in \texttt{reach}(P'),\ \sigma.c_{loc}(l_s) \leq\ M$$

Then M is also a bound of every reachable local counter at l_s in P:

$$\forall \sigma \in \texttt{reach}(P),\ \sigma.c_{loc}(l_s) \leq\ M.$$

[Coq Proof]

Theorem 5 is stated in terms of local execution counters, but the exact same property is valid when considering global execution counters as well. That is, $\sigma.c_{\text{loc}}$ can be replaced with $\sigma.c_{\text{glob}}$ in Theorem 5. This more general theorem, considering both sets, is the main theorem of slicing correctness with respect to execution counters.

The second correctness theorem related to program slicing states that *slicing a terminating program preserves termination*. Formally stated as Theorem 6 below, this property is quite specific to our loop bound estimation method and seldom mentioned in discussions about program slicing. As a fact, most slicing applications do not actually need it, and standard slicing algorithms (such as the one presented in Section 5.2) do not guarantee it. Our *termination-preserving* slicing algorithm, described in Section 5.4.1, will however take it into account.

Theorem 6 (Program Slicing Preserves Termination)**.**
Let P be a program and l_s a program point of P.
Let P' be the program slice of P with respect to slicing criterion l_s.
If P terminates, then P' terminates.

[Coq Proof]

This theorem does not mention execution counters since they are already taken care of by Theorem 5. The main intuition behind this termination theorem is given by its contrapositive: if a program slice does *not* terminate, then we know the original program does not terminate either. Since one of our hypotheses is that the original program must terminate, this entails that non-terminating slices do not need to be addressed: if a slice is non-terminating, then the original program was also non-terminating, and in this case we have nothing to prove. We can summarize this theorem as *program slicing does not introduce divergence*.

5.4.1 Computing and Proving a Termination-Preserving Program Slice

Since the correctness theorems related to program slicing for the loop bound analysis require additional properties on the computed slice, we need to modify our program slicer to take them into account.

Our *slice set calculator* is mostly unchanged; everything related to the computation of dependencies remains the same. The only change is related to the `Extra` information used by the checker: besides the `RV`, `NObs` and `DObs` sets, we add an extra set `DExit` containing *exit distances*, which are equivalent to *next observable distances*, but for vertices after the slice. Their purpose is made clear in the next section.

From Slice Set to *Terminating* Program Slice

Our previous slice builder was only concerned with preserving behavior up to the slicing criterion. Our new requirements for the program slice require the slice builder to extend its reasoning to vertices after the slice. Fortunately, this change is trivial and emulates the same method already applied to avoid infinite loops inside the slice: all we need to do is to find a *unique*, *reachable* node such that all program points after the slice can reach it in a finite number of steps. Thanks to our CFG normalization, this vertex always exists and it is the *exit vertex* l_{exit}.

We use an *exit distance* equivalent to the next observable distance to decide which branch to choose. This information is provided by the `DExit` function we just added to our slice set calculator. An example of its usage is given in Figure 5.10.

Note that most slicing algorithms do not consider this situation at all, since they are not concerned with program termination. For instance, when debugging, we do not take into account what happens after the slice, so a program slicer can make an arbitrary choice in this case. For this reason, this special treatment is hardly mentioned in proofs of program slicing, despite being a somewhat elegant counterpart to the *next observable distance*.

Proving a Terminating Program Slice

A new proof is required for the terminating program slicer, and the modifications it imposes are more significant than those performed on the program slicer. Indeed, since our semantics has changed, our simulation proof needs invariants relating execution counters and termination issues, and this in turn entails extra properties in the axiomatization. In this section, we exhibit a new simulation relation between original and sliced programs, which takes into account execution counters and preservation of termination. Finally, we present a more technical and detailed simulation proof, which corresponds closely to the Coq formalization of program slicing and that contains some interesting aspects related to termination.

Detailed Simulation Proof

The simulation between states used in the soundness proof of program slicing is a *weak simulation*, i.e. it does not preserve infinite executions. In this section, we present the new matching states relation, then we briefly justify why our program slicer does not preserve infinite executions. We then detail the weak simulation, outlining the proof of an equivalent version of Lemma 4 that is adapted to the instrumented semantics.

Matching States Relation The matching states relation in Figure 5.11 needs to be modified for the instrumented semantics of the loop bound analysis. Figure 5.12 presents the new matching relation, with updated rules (R_1), (R_2) and (R_3).

$$\frac{l \in SL(l_s) \quad E \simeq_{RV(l)} E' \quad c(l_s) = c'(l_s)}{(l, E, c) \sim (l, E', c')} \ (R_1)$$

$$\frac{l \notin SL(l_s) \quad l' \notin SL(l_s) \quad \texttt{NObs}(l) = \texttt{NObs}(l') \quad E \simeq_{RV(l)} E' \quad c(l_s) = c'(l_s)}{(l, E, c) \sim (l', E', c')} \ (R_2)$$

$$\frac{l \notin \texttt{dom}(\texttt{NObs}) \quad c(l_s) = c'(l_s)}{(l, E, c) \sim (l_{\text{exit}}, E', c')} \ (R_3)$$

Figure 5.12: Matching between execution states of a program and one of its slices (having $SL(l_s)$ as its slice set) on the instrumented semantics with execution counters.

In Figure 5.12, execution states are now presented as triples (l, E, c), where c denotes either a local or a global counter. The actual states contain one of each set of counters, but since they are

symmetrical, we present only one of them on the matching rules. c can be replaced by c_{glob} or c_{loc} on any of these rules.

All rules now state the equivalence of execution counters for the slicing criterion. Other than that, rules (R_1) and (R_2) are identical to their previous counterparts. Rule (R_3) matches any state of the initial program *after the slice* (for which there are no next observable vertices) with the state of the program slice at program point l_{exit}. Reaching this point is important to ensure termination.

Program Slicing and Preservation of Termination This section presents a detailed discussion about preservation of terminating behaviors, with examples illustrating each possibility.

We can classify program slicing algorithms according to two different criteria regarding termination:

- preservation of terminating behaviors: if a program P terminates, then its slice P' also terminates;
- preservation of non-terminating behaviors: if a program P does not terminate, then its slice P' also does not terminate.

The program slicer we consider preserves terminating behaviors, such as needed to prove Theorem 6, but does not preserve non-terminating behaviors. This improves precision, since infinite loops may be sliced away, resulting in smaller (more precise) slices. In other words, if a program terminates, then its slice terminates, but if a program does not terminate, its slice may or may not terminate.

To see why it is preferable to use a program slicer that does not preserve non-termination, let us consider the following C program:

```
i = 0; while (i < 5) { i++ }; j = 1
```

We can clearly see this program always terminates. We can also see that the loop has no impact on the value of variable j at the last program point. Therefore, when slicing this program with respect to to `j = 1`, we would expect it to be reduced to the following program:

```
j = 1
```

However, our slicer does not have any information about dynamic variable values. It operates on a syntactic level, with information given by the CFG. A correct slice requires a conservative approach, which means that in the previous example it must consider the last statement `j = 1` as control-dependent on the `while` condition. Indeed, the execution of `j = 1` depends on the result of the loop condition: if the loop never terminates, then `j = 1` is never executed. Otherwise, `j = 1` is executed once. This means, in practice, that a program slicing algorithm that preserves non-termination can never slice a loop away. A slicer that preserves such non-termination needs to consider a different kind of control dependency, called *weak control dependency*. Using this kind of dependency corresponds to adding an arc between each loop condition vertex and the program points after the loop, to represent the dependency that these vertices have on the loop execution. The slices produced this way are larger (less precise), negatively impacting the analyses that use them. They are seldom used in practice.

We have chosen a slicing algorithm that does not preserve non-termination, and that can slice away loops unrelated to the slicing criterion. For instance, consider the following C program:

```
i = 0; while (i == 0) { /* nop */ }; i = 1
```

This program contains an obvious infinite loop and never reaches the last statement (`i = 1`). Yet, slicing it with respect to this last statement yields a program equivalent to:

```
i = 1
```

This is not a problem, since we do not state any properties about infinite executions.

Weak Simulation We now proceed to describe the simulation diagram used in our proof. For the weak simulation, we consider two matching states, $\sigma_1 \in \texttt{reach}(P)$ and $\sigma'_1 \in \texttt{reach}(P')$: $\sigma_1 \sim \sigma'_1$. An execution step $\sigma_1 \to \sigma_2$ is called *observable* if $\sigma_1.l \in \texttt{SL}(l_s)$ (the departure vertex is in the slice set), and *silent* otherwise. A silent step corresponds to an execution step from a no-op or a constant predicate in the sliced program.

The diagram in Figure 5.13 presents the two possible configurations for the simulation: Figure 5.13a depicts an observable step in P, which corresponds to exactly one step in P', while

Figure 5.13b depicts a silent step in P, which corresponds to any number of steps (including zero) in P'.

(a) Observable step (b) Silent step

Figure 5.13: Simulation diagram for the weak simulation used to prove correctness of the slicing. Continuous lines represent hypotheses, dashed lines represent conclusions.

The horizontal line between σ_1 and σ'_1, as well as the arrow from σ_1 to σ_2, are the hypothesis of the simulation. One must prove the existence of a state σ'_2, an arbitrary number of steps from σ'_1 to σ'_2 (dashed arrow between these states) and the matching relation between σ_2 and σ'_2.

As a side note, we remark that strong simulation diagrams of the type shown in Figure 5.13b cannot be used for semantics which are capable of distinguishing several kinds of infinite behaviors. This is mentioned as the *stuttering* problem [23], and the solution is to find a *measure* for states that decreases at each step. For a weak simulation, however, such infinite behaviors are indistinguishable, therefore no such measures are necessary.

Matching States Step-by-step We now detail the simulation step by step, using the matching rules from Figure 5.12. We present it under the form of an algorithm. For simplicity, we replace the notations of the program points, using indices instead of the states: l_1 for $\sigma_1.l$, l'_1 for $\sigma'_1.l$, and so on.

Assume program P satisfies the constraints (C_1) to (C_{10}). Its entry point l_{entry} is always in the slice set[8], so at the beginning of the execution, rule (R_1) holds. The first step is always an observable step $\sigma_1 \to \sigma_2$. P' follows in lock-step ($\sigma'_1 \to \sigma'_2$) to the same vertex l_2, which corresponds to the intuitive idea that a statement in the slice set is executed in both programs.

After the step, if $l_2 \in \texttt{SL}(l_s)$, then rule (R_1) still holds. Otherwise, either l_2 has a next observable vertex (and rule (R_2) holds), or it does not ($l_2 \notin \texttt{dom}(\texttt{NObs})$) but rule (R_3) holds.

A silent step $\sigma_1 \to \sigma_2$ occurs when $l_1 \notin \texttt{SL}(l_s)$. In this case, as indicated in Figure 5.13b, the sliced program can perform any number of steps. There are two different cases here, depending on whether l_2 is in the slice set. An interesting situation arrives when $l_2 \notin \texttt{SL}(l_s)$: in this case, the sliced program does not perform any steps. This causes a *desynchronization* of the programs, since they are now in different program points. This is the crux of the simulation: rule (R_2) holds in this case, and will hold as long as P does not reach a vertex in the slice set. Properties (C_2) to (C_4) ensure that no relevant variable will be modified in P. Figure 5.14 presents an example of some steps performed during the simulation, with the original program on the left and the program slice on the right.

When the execution returns to the slice set ($l_2 \in \texttt{SL}(l_s)$), the programs can be resynchronized. In this case, the sliced program performs one or several (silent) steps ($\sigma'_1 \to^+ \sigma'_2$) until it reaches $l_2 = l'_2$, where l_2 (respectively l'_2) is the next observable vertex of l_1 (respectively l'_1). This is where the *next observable distance* comes into play: it exhibits a finite number of steps that are required for the resynchronization to happen. After resynchronization, both states match again and rule (R_1) holds.

This alternation between vertices inside and outside the slice set can happen several times, until either P reaches the exit node (if it is in the slice set) and the simulation ends, or until P reaches a vertex that is after the slice ($l_2 \notin \texttt{dom}(\texttt{NObs})$). In this case, property (C_7) ensures that we cannot return to the slice set anymore. P' then performs an arbitrary number of steps until it reaches the

[8] $l_{\text{entry}} \in \texttt{SL}(l_s)$ is an enforced property on $\texttt{SL}(l_s)$. It is not listed in Figure 5.9 because it is specific to our normalized CFGs, but not necessary in the general case.

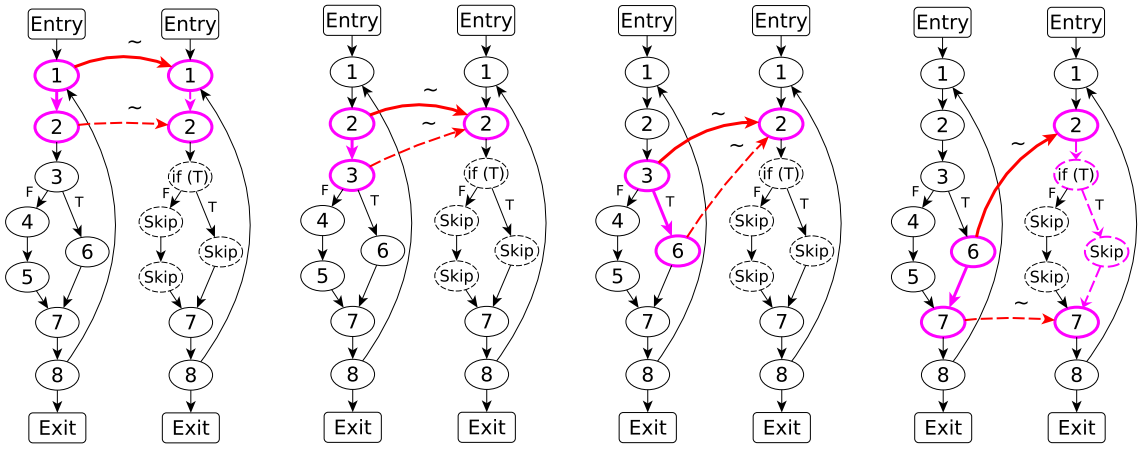


Figure 5.14: Example of the evolution of a simulation between a program (left) and its slice (right). Thick solid arcs represent hypotheses, dashed arcs represent conclusions.

(unique) end vertex l_{exit}. By using the *exit distance*, we know that l_{exit} can always be reached in a finite number of steps. This ensures termination of P'.

Afterwards, states match by rule (R_3), for any further steps performed by P. We do not need to match match relevant variables anymore.

This concludes the proof of the modified Lemma 4[9], adapted to the instrumented semantics, and with it the correctness theorems of program slicing applied to the loop bound analysis.

5.5 Conclusion

Our approach to computing and validating program slices, mixing both trusted components and *a posteriori* validation of complex computations, constitutes a loosely coupled module with a reusable part (computation of the slice, plus Theorem 4 for general slicing correctness) and a specialized component developed for the correctness of a loop bound analysis (Theorem 5 and Theorem 6). It has been successfully integrated into the loop bound estimation framework.

The extra constraints imposed on the slicing by the loop bound analysis have shown that it is possible to adapt the proof to a different semantics, while leaving most of the computation untouched. They have also shown that there is always some coupling between different components of a multi-stage analysis. It is inevitable, but this is also one of the strengths of a mechanically verified proof: it is easy to fall into the trap of integrating components without closely looking into their interactions, risking the introduction of subtle bugs. Proving the software correct ensures these seemingly minor details are taken into account at all times.

Overall, the formally verified program slicer presented here is another example of a successful integration of *a posteriori* validation and standard verification. Delegating the most complex algorithms to the validator minimizes proof effort, and verifying the result of the validator ensures high confidence in its result.

[9]The Coq proof of this modified lemma is available on the online development.

Chapter 6

Value Analysis for C

Value analysis is one of the components of our verified WCET estimation tool. More generally, value analyses are useful in several contexts: program verification, static analysis, compiler optimization. We consider here the broader context of safety-critical software development. Implementing a value analysis for a language like C is a complex task, with several kinds of trade-offs between efficiency, precision and completeness. In this chapter, we describe the development and evaluation of a value analysis for CompCert's Cfg language, an intermediate step in the development of a full-fledged, mechanically verified, value analysis for the C language. The work in this thesis focuses on the experimental evaluation, but a presentation of the analyzer is necessary to understand some of the issues and challenges in the evaluation.

6.1 Motivation

Static analyzers are fundamental tools for the development of safety-critical embedded software. Testing all possible program inputs is not feasible, and manually inspecting code is not a scalable approach for the ever-increasing size[1] and complexity of software present in safety-critical systems. Ideally, the code should be automatically verified at the source level (which is the one closest to the system specifications), and then compiled using a verified compiler which does not introduce any bugs. This would improve confidence in the software and avoid costly manual inspections of the assembly code during the process of software certification.

As presented in the introductory context about formal verification (Section 2.2.2), the CompCert C compiler preserves the semantics of *safe* programs, that is, programs without undefined behaviors. Unfortunately, in C many programs present undefined behaviors. This can be due to "standard" programming errors, such as violation of array bounds or use of uninitialized local variables. Worse, it can be due to situations which most programmers do not even recognize as incorrect [22], such as the use of signed overflow or comparisons between invalid pointers. These errors are more severe because programmers inadvertently rely on default behaviors of compilers, which are in fact unspecified in the C semantics, such as assuming two's complement arithmetic for signed overflows. Such errors go unnoticed until the compiler changes its behavior, at which point they are harder to identify.

It is not possible to devise a program to automatically decide whether any C program has only defined behaviors. However, it is possible to decide that *some* C programs do not have any undefined behaviors. By restricting the kinds of programs to be analyzed (e.g. forbidding function pointers) and by devising precise static analyzers, we can rule out the possibility of undefined behavior for these programs.

The objective of the French Verasco research project is to develop a static analyzer for embedded C, similar to the one developed in the Astrée project [18], but machine-checked in Coq. This analyzer uses CompCert's formal semantics for the C language to check whether a given program

[1] As an example, between the Boeing 777, which entered into service in 1995, and the newer Boeing 787, whose service entry took place in 2011, there is an 8-fold increase in the number of lines of code [63].

has a defined semantics. If so, compiling this program with CompCert will yield an assembly program which is formally verified to be free of runtime errors.

The main component of this project is an abstract interpretation-based value analysis for the C language. We describe the first version of this value analysis and its implementation in Section 6.2, as a prerequisite to the evaluation process and experimental results presented in Section 6.3, which is related to the work in this thesis.

6.2 Verasco's Value Analyzer (VVA)

Verasco's Value Analyzer (abbreviated as *VVA*) is based on a modular architecture of abstract domains [11], ranging from simple numeric domains (such as intervals) to memory models of C code including dynamic allocation. The overall architecture is presented in Figure 6.1. The layered approach allows parallel development and exchanging of different versions of a module, enabling trade-offs between precision and efficiency.

VVA's development operates in an iterative manner. Instead of defining and proving the entire development in a single step, it proceeds by incrementally adding or improving features. To avoid spending time with unnecessary proofs, each new iteration is first tested and evaluated (using code generated from the Coq development), and proved correct only if experimental results are satisfactory. The final program takes a C file as input, compiles it (using CompCert) down to the Cfg language, and then applies the value analysis.

The value analyzer uses an abstract interpretation framework. Most concepts about abstract interpretation have been presented in Section 3.2. We focus here on the aspects specific to its implementation, detailing its abstract domains, fixpoint iterator and transfer function. In this section, we present the analyzer from a bottom-up approach (from the simplest component to the most complex one), as indicated in Figure 6.1. We start with the definition of an abstract domain, using a numerical domain as example. Then we present the product of different abstract domains. From numerical domains, we obtain abstract numerical environments. Afterwards, we describe the C memory model used in the analysis, then the fixpoint iterator, and finally the transfer function.

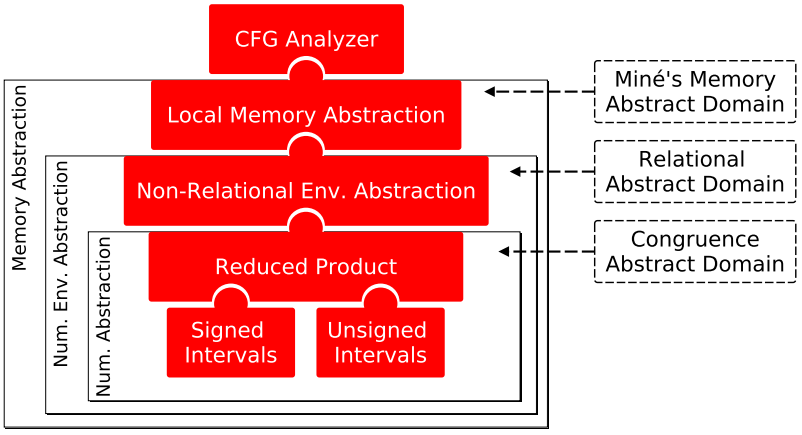


Figure 6.1: Design of the abstract domains of the VVA value analysis. The red pieces correspond to the domains present in the current version. Future directions of evolution are presented on the right.

6.2.1 Numerical Abstract Domains

A numerical abstract domain $\mathcal{A}dom$ in our formalization is a tuple $(\mathcal{A}, \mathcal{B}, \gamma, \leq, \sqcup, \top, \nabla)$.

$\mathcal{A}$ is the type of *abstract values* of our domain. For instance, $\mathcal{A}$ can be the set of intervals of machine integers 32-bit long, which we call `Itv`. Values of this domain include $[0, 3]$, $[2, 2]$ and $[-\infty, +\infty]$, used to represent the interval containing every possible 32-bit signed value, that is, $[-2^{31}, 2^{31} - 1]$.

$\mathcal{B}$ is the type of *concrete values* of our domain. In our analysis, this will often be the domain of machine integers (32-bit long) `Int`.

$\gamma : \mathcal{A} \to \mathcal{P}(\mathcal{B})$ is a *concretization function*, which maps an abstract value to a set of concrete values. For instance, for $\gamma : \texttt{Itv} \to \mathcal{P}(\texttt{Int})$, we have $\gamma([-2, 3]) = \{-2, -1, 0, 1, 2, 3\}$.

$\leq$: $\mathcal{A} \times \mathcal{A}$ is a partial order relation for abstract values. This is the abstract counterpart of logical implication. $a \leq b$ means the information a is more precise than b. It is often defined as the set inclusion relation between concrete values. For instance, we say that $[1, 3] \leq [0, 5]$, because all values in $[1, 3]$ are included in $[0, 5]$. Reciprocally, $[1, 3] \not\leq [2, 5]$, since $1 \notin [2, 5]$ and $4 \notin [1, 3]$, so we cannot compare these intervals.

$\sqcup : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is the *lowest upper bound* operator (also called *join*). It computes the smallest abstract value which is greater than its operands: $a \leq a \sqcup b$ and $b \leq a \sqcup b$. In general, when proving soundness we do not need to prove that it is indeed the *lowest* one, simply that it is an upper bound. It is often defined as the set union of the concrete values. In our examples, $a \sqcup b$ is the smallest (contiguous) interval containing a and b. For instance, $[-1, 0] \sqcup [4, 5] = [-1, 5]$.

$\top$, called *top*, is the greatest element of the abstract domain. For any $a \in \mathcal{A}$, $a \leq \top$. In our `Itv` domain, it corresponds to the interval $[-\infty, +\infty]$ containing all 32-bit machine integers.

$\nabla : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a *widening* operator, similar to $\sqcup$ but intended to accelerate convergence during fixpoint iteration by performing an over-approximation of the upper bound. As we will see, ∇ is not necessary for correctness, but in practice it is important for the efficiency and sometimes termination of the analysis. A very simple (and imprecise) definition for ∇ might be: $a \nabla b = \top$ for any a and b. This mainly forces the convergence in one step, but it is extremely imprecise. More sophisticated versions include doubling the size of the domain at each step and widening based on constants in the program (e.g. is there is a condition $i < N$, then one widening step might go from $[0, 1]$ to $[0, N]$).

Each operator of an abstract domain must verify a few properties, which are used during the proof of correctness of the value analysis. Traditionally, in abstract interpretation there is a large set of properties which are demonstrated for the operators, such as the presence of a *Galois connection* between the abstract and concrete values, the fact that the join operator produces a *least* upper bound, that the widening operator is stable, etc. However, the development of VVA is based on a lightweight approach that minimizes proof effort as long as correctness is guaranteed. We avoid proving properties that are only related to completeness or efficiency. As we will see later, these are taken into account through testing and performance evaluation.

The fundamental properties for the correctness of the abstract domain are:

1. Monotonicity of the concretization function: an element of the abstract domain which is larger (less precise) than another must contain at least all of the former's concrete values.
   $\forall a_1, a_2 \in \mathcal{A}, a_1 \leq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2)$

2. Soundness of the upper bound: the upper bound of two abstract elements must contain (at least) every one of their concrete values.
   $\forall a_1, a_2 \in \mathcal{A}, (\gamma(a_1) \cup \gamma(a_2)) \subseteq \gamma(a_1 \sqcup a_2)$

3. Soundness of the $\top$ operator: the concretization of the *top* element must contain every possible concrete value.
   $\forall b \in \mathcal{B}, b \in \gamma(\top)$

All three properties are related to the notion of over-approximation as correctness of the analysis: to larger abstract values correspond larger sets of concrete values. The $\leq$ relation and the $\sqcup$ and $\top$ operators must be compatible with this notion for the analysis to be correct. Note that the widening operator is not mentioned here, because its correctness is validated *a posteriori* after fixpoint iteration (Section 6.2.5).

6.2.2 Domain Products

The use of *direct products* allows the combination of several domains into a single, more precise domain for the analysis. The underlying idea is that defining and proving a single universal abstract

domain is very costly; it is more efficient to define a set of several specialized domains which exchange information between them, each one capturing a specific property of the concrete values.

Two abstract domains $\mathcal{A}$ and $\mathcal{A}'$ combined in a direct product have abstract values $(a, a') \in \mathcal{A} \times \mathcal{A}'$ and a concretization function $\gamma_{\text{prod}}((a, a')) = \gamma(a) \cap \gamma'(a')$ that is the *intersection* of the values in each domain. This is especially useful when a domain has a particular weakness that another domain handles better.

For instance, small sets and intervals can be combined to represent sparse sets, ensuring precision when there are few elements in the abstract value (using the small set), and scalability when there are several (using the interval). For instance, the small set $\{1, 2, 4, 5\}$ is more precise than the interval $[1, 5]$, but also more costly in terms of memory and processing, since each value is enumerated explicitly.

Because the concretization of a domain product corresponds to the *intersection* of both domains, the final result can only improve in precision. The price to pay is an increase in complexity and development effort. First, it is necessary to prove each domain correct in isolation; if there are two domains instead of one, then this effectively doubles the effort. Then, it is necessary to define and prove the combination of results, that is, the product itself. This requires a few more correctness proofs.

As example of a direct product, we can cite the use of *signed* and *unsigned* integer intervals. In LLVM[2] and CompCert, some intermediate representations do not distinguish between signed and unsigned integer pseudo-registers (one of the reasons being to simplify and minimize the number of operators and value representations). For a static analysis, however, the loss of signedness information decreases its precision. To compensate for it, analyses such as the one developed by Navas *et al.* [54], as well as Verasco's, use special interval representations to recover this information.

In the case of Verasco's value analysis, this is done via a direct product of both signed and unsigned intervals. As stated before, this product uses the intersection of both domains, avoiding precision losses when a program uses values which overflow or that are not representable by one of the intervals, such as those in the program in Figure 6.2. In this program, using only signed or only unsigned intervals entails a loss of precision, due to an overflow either in the first or in the second `if`. The graphical representation of the interval bars in the figure allows for a visual comparison of the differences, which we detail in the next paragraphs.

In Figure 6.2, we have a signed integer `s` and an unsigned integer `u`. A non-deterministic choice (represented by `if (*)`) assigns a large value for `u`, either $2^{31} - 1$ or its successor, which are both in the middle of the representable domain of an unsigned variable. After the `if`, the domain of variable `u` is precisely represented by the interval $[2^{31} - 1, 2^{31}]$, which contains two elements. However, if we try to represent if using a *signed interval*, due to overflow of 2^{31}, which is not representable in a 32-bit signed variable, the actual interval will be $[-2^{31}, 2^{31} - 1]$, that is, $\top$, the interval containing every possible signed value.

The following line in the program is another non-deterministic choice, this time operating on the signed variable `s`. The signed interval $[-1, 0]$ is a precise representation of its domain, located in the middle of the domain of signed integers. Using an unsigned interval here incurs a loss of precision equivalent to the one in the previous line, when using a signed interval: instead of a precise interval containing two values, we end up with the $\top$ interval.

Finally, in the last line, `u + (s - 1)` operates on both signed and unsigned variables. Without a product of abstract domains, we are forced to make a choice, either picking a signed interval for the result, or an unsigned interval (we have a choice here because CompCert's semantics for integer overflow is well-defined). In both cases, we obtain the *top* interval due to imprecision propagated from one of the previous lines. Using the product of both signed and unsigned intervals, however, we manage to maintain a precise interval at each line, obtaining in the end the precise interval $[2^{31} - 3, 2^{31} - 1]$.

[2]LLVM, formerly *Low Level Virtual Machine*, is a compiler infrastructure including modular analyses and an intermediate representation which is compilable to several architectures. LLVM also includes the Clang C compiler, an LLVM front-end for C.

```
  int main(void) {
    signed s;
    unsigned u;
1   if (*) u = 2^31 - 1; else u = 2^31; // u ∈ [2^31 - 1, 2^31]
2   if (*) s = 0; else s = -1; // s ∈ [-1, 0]
3   return u + (s - 1); // res ∈ [2^31 - 3, 2^31 - 1]
  }
```

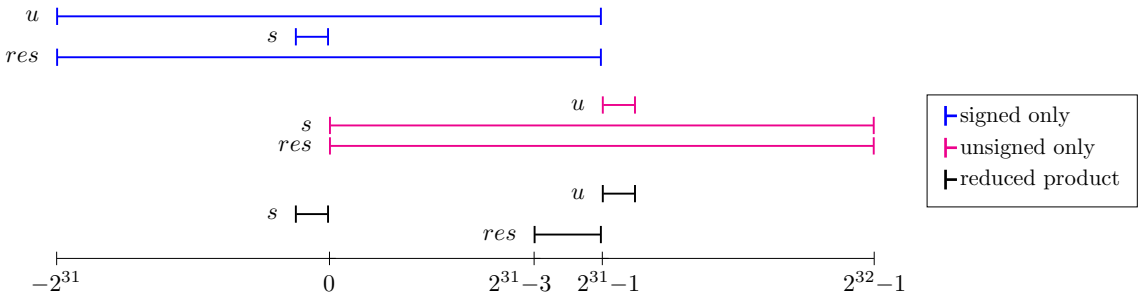


Figure 6.2: Program demonstrating gain in precision through the use of a product of signed and unsigned intervals, followed by a graphical representation of these intervals. The precise solution for variable *res* after line 3 is the interval $[2^{31}-3, 2^{31}-1]$, obtained with the reduced product.

6.2.3 Abstract Numerical Environments

The second layer in the modular domain hierarchy corresponds to abstract numerical environments of program variables. These environments require abstract domains where $\mathcal{B} = \mathcal{P}(\mathcal{V}ar \to \mathcal{N}um)$, that is, the concretization maps variables to numerical values (integers or floats).

To avoid losing precision, abstract environments operate on *numerical expression trees*, of type *nexpr*, which are defined as follows:

$$e_{\text{tr}} ::= \text{NEvar } id \mid \text{NEconst } c \mid \text{NEunop } op_1\ e_{\text{tr}} \mid \text{NEbop } op_2\ e_{\text{tr}}\ e_{\text{tr}} \mid \text{NEcond } e_{\text{tr}}\ e_{\text{tr}}\ e_{\text{tr}}$$

A numerical expression is either a variable (NEvar *id*), a numerical constant (NEconst *c*), an unary operation (NEunop op_1 e_{tr}), a binary operation (NEbop op_2 e_{tr} e_{tr}) or a conditional expression, similar to C's ternary operator `?:` (NEcond e_{tr} e_{tr} e_{tr}). As an example, `x + 1 ? y : -2` is represented by:

(NEcond (NEbop *plus* (NEvar x) (NEconst 1)) (NEvar y) (NEunop *unary-minus* (NEconst 2)))

Note that these expressions correspond to those of the Cfg language. They are also equivalent to those in the C code, so that the analysis can be adapted to any higher-level language. The benefit of having such expression trees is even greater when the environments do not use relational abstractions. In this case, the expression trees allow the propagation of some relational information from one variable in the expression to another.

As an example of a benefit provided by expression trees, let us consider the expression `if (x < y && y < 10 && z < x)`. Analyzing this expression, we infer that $x < 9$ and $z < 8$. However, in the absence of expression trees (e.g. as in 3-address code), the conditional is broken into a sequence of simpler conditions:

```
if (x < y) { if (y < 10) { if (z < x) { /* all true */ } else ...
```

Because the conditions are now scattered among different program points, evaluation is performed locally for each condition. When `x < y` is evaluated, since we have no bounds on either x or y, the only information obtained from this condition is that $x < 2^{31} - 1$ and $y > -2^{31}$. Afterwards, when `y < 10` is evaluated, we learn new information about y, but since the abstraction is non-relational, it is too late to propagate this information back to variable x. Without expression trees, we lose the ability to propagate information from one sub-condition to another.

The interface of abstract environments contains three operators, parameterized by the type $\mathcal{A}$ of the abstract domain:

- `range`: $nexpr \to \mathcal{A} \to$ `Itv` returns the numerical range of an expression (an interval);
- `assign`: $\mathcal{V}ar \to nexpr \to \mathcal{A} \to \mathcal{A}$ corresponds to the assignment of a numerical expression to a variable;
- `assume`: $nexpr \to \mathcal{A} \to \mathcal{A}$ assumes that a numerical expression evaluates to a non-zero value.

The `range` operator is used to retrieve the result of the analysis. The other two operators, `assign` and `assume`, perform the actual propagation of the analysis, guided by the fixpoint iterator (explained later). At each program point, if the associated statement is an assignment, then `assign` propagates the right-hand side expression to the variable in the left-hand side. If the program point corresponds to a condition, `assume` modifies the environment to include the fact that the conditional expression is true in the `true` branch, and that its negation is true in the `false` branch. The `assume` operator also allows *backward propagation* of information, from the branches to the condition. This is what allows the previous example, `if (x < y && y < 10 && z < x)`, to obtain precise information.

The instantiation of an abstract environment includes the definition of a few extra operators, such as a *meet* operator (lower bound) for the abstract domain, as well as forward and backward propagation for unary and binary operators. Forward propagation uses information about inputs to provide information about the output of the operation. Backward propagation uses information about the expected result of the operation to refine information about its inputs. Combining both is fundamental for precision.

Proof of soundness for all operators is required for the correctness proof of the value analysis. As in the previous case, correctness is related to the concretization of the abstract environments.

6.2.4 Abstract Memory Domain

The previous environment domain is still unaware of the C memory model. For instance, it has no knowledge about pointers and memory blocks. The third layer in the domain hierarchy corresponds to the abstract memory model, `mem_dom`, which is similar to the abstract environment but has the following commands: `Assign` and `Assume`, like before, but also `Forget` and `Store`.

`Forget`: $\mathcal{V}ar \to \mathcal{A} \to \mathcal{A}$ is used when incertitude about a variable introduces an over-approximation of its value. In other words, we lose all useful information about this variable, reverting it to $\top$.

`Store`: $\kappa \to Mem \to \mathcal{A} \to \mathcal{A}$ corresponds to a memory store. It allows modification of values stored in memory.

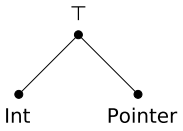
Abstract memory models concretize to $\mathcal{B} = \mathcal{P}(Env \times Mem)$, where Env is the environment (mapping of variables to values) and Mem is the memory. The concretization function, defined below, reads as follows: the concretization γ_{Mem} of the abstract memory domain is a function whose input is an element of the abstract domain ab, of type $\mathcal{A}$, and whose output is the set of pairs (e, m) (of abstract environments and memory states) such that there exists at least one numerical environment in the concretization $\gamma(ab)$ that corresponds to e for numerical values and pointer offsets. Note that we do not track any information about the contents of the memory, since m is not constrained in the formula.

$$\gamma_{Mem}(ab : \mathcal{A}) :=$$
$$\{(e, m) | \exists \rho, \rho \in \gamma(ab) \ \wedge$$
$$\forall x \in \mathcal{V}ar, \forall i \in \texttt{Int}, (e[x] = Vint(i) \ \vee \ \exists b, e[x] = Vptr(b, i)) \to \rho(x) = i\}$$

Intuitively, this definition for the concretization function is ensuring that, whether the variable corresponds to an integer or to a pointer offset, both cases are correctly taken into account. We explain below that the need of considering both integers and pointer offsets is related to typing ambiguities, and why they happen in the value analysis.

Typing Ambiguities In the C standard, there is a difference between *integer variables* (which include characters, integers and boolean variables) and *pointer variables* (which contain addresses for references). In the Cfg value analysis, however, such information may be lost between the different abstraction layers. Typing information helps maintain this information.

At the lowest abstraction level, where numerical expressions are dealt with, expressions such as unsigned equalities `x ==u y` are converted to simple numerical expressions such as `x == y` before being used by the `assume` operator. A naive translation of the result ot this operator might lead to the conclusion that a variable $x = Vint(0)$ is equal to a variable $y = Vptr(b, 0)$, for instance, while in the original unsigned comparison in C this would never happen. To avoid such situations, the analysis computes some extra type information using a simple lattice.



A simple type analysis using this lattice results in one of these three possibilities: a given variable (1) *must have* type `Int`, or (2) it *must have* type `Pointer`, or (3) its type *may be* either of those ($\top$). Situations (1) and (2) are the norm in most programs. With this typing information, the different layers of the value analysis communicate efficiently while avoiding incorrect conclusions.

The commands of the memory domain use their corresponding operations on the abstract environment (except for `Forget`, which can be implemented by an `assign` $\top$ operation). These, in turn, use the numerical domains. The value analysis for the Cfg language is a combination of this memory domain with the fixpoint iterator defined in the next section.

6.2.5 Fixpoint Iterator

The fixpoint iterator is based on Bourdoncle's algorithm [13]. The iterator behaves as follows: given a program's control flow graph and the transfer function of the analysis, the iterator starts with the *bottom* value of the abstract domain, repeatedly applying the transfer function to the abstract domain until it reaches a fixpoint. It then returns this fixpoint as final result.

One of the strengths of Bourdoncle's algorithm is that, during the iteration of the transfer function, it follows the loop nesting structure of the CFG: program points inside loops are processed before program points outside of the loop. This minimizes the number of recomputations of the transfer function necessary for stabilization. This also helps when dealing with the widening operator: applying it in a different order may cause a loss of precision.

Widening incorporates different strategies and is partially configurable by the user, allowing various trade-offs between efficiency and precision. For instance, it can be configured to use integer constants in the source code as widening hints, to avoid excessive widening. This may increase the analysis time, however. Besides widening, the fixpoint iterator also uses *decreasing iterations* to improve precision. They work similarly to *narrowing* operators but are easier to implement. All these aspects correspond to different *heuristics*, useful for efficiency and precision of the result, but only indirectly related to the correctness of the analysis. Instead of proving all of them correct, which is a costly effort, a more efficient solution is attained by applying *a posteriori* validation. It leverages the fact that the process of obtaining the fixpoint is not directly related to the correctness of the final result. The only property necessary for correctness is that *the result of the fixpoint iteration must be a post-fixpoint of the transfer function.* Note that it can be *any* post-fixpoint, not necessarily the least one, and that the steps performed to reach this fixpoint are unimportant[3]. Besides, there is an efficient way to test if a given candidate is indeed a (post-)fixpoint: we apply the transfer function on the candidate and check if the new result is smaller (in the order given by the abstract lattice) than the previous one. In terms of efficiency, the computation of a single extra step of the transfer function has a negligible cost. All these aspects lead to an efficient *a posteriori* validation of the result of the analysis. The validated fixpoint iterator is a short function:

[3]The lattice fixpoint diagram, previously presented in Figure 3.7, illustrates this fact.

```
let transf (stmt : Stmt) : list(𝒫𝒫 × (𝒜_Mem → 𝒜_Mem)) =
  match stmt with
    | skip(s) → [(s, λ(ab).ab)]
    | assign(x,e,s) → [(s, Assign(x,e))]
    | store(chunk,e,a,s) → [(s, Store(chunk,e,a))]
    | if(e,l_true,l_false) → [(l_true, Assume(e)), (l_false, Assume(¬e))]
    | return → []
```

Figure 6.3: Transfer function of the value analysis.

```
compute_fixpoint (ab: 𝒜dom) (transf: Stmt → list(𝒫𝒫 × (𝒜_Mem → 𝒜_Mem)))
                 (cfg: CFG) (l_entry: 𝒫𝒫) (init: 𝒜_Mem) {
  /* fxp is a candidate fixpoint */
  fxp = calc_external_fxp(ab,transf,cfg,l_entry,init)
  if check_fxp(ab,transf,cfg,l_entry,init,fxp) { return fxp }
  else { return ⊤ }
}
```

The `compute_fixpoint` function receives as input an element of the abstract domain (`ab`), the transfer function (`transf`, detailed in the next section), the program's CFG (`cfg`), the initial program point (l_{entry}) and the initial abstract value (`init`). All of these arguments are passed on to the non-proved `calc_external_fxp`, which is where the actual fixpoint is computed. No proof is required for this code. The result is checked by `check_fxp` (which reuses most arguments to recompute a single step of the transfer function), and returned if it is indeed a fixpoint. Otherwise, a safe over-approximation ($\top$) is returned instead. In particular, we do not need to prove the termination of the fixpoint iterator. The overhead associated with this validation is minimal.

6.2.6 Transfer Function

The transfer function, whose input is the current statement to be interpreted, produces a list of *abstract state modifier functions* for each possible successor of the current statement. That is, the transfer function does not directly apply the modification to the abstract value, but it produces the command that the fixpoint iterator will apply when the program point is reached. The pseudocode of the transfer function is given in Figure 6.3. For the *skip* statement, the analysis applies the identity function: for any abstract value *ab* given to the fixpoint iterator, it will return the same abstract value. The *Assign*, *Store* and *Assume* functions are those of the abstract memory model, detailed in Section 6.2.4.

The transfer function is given to the fixpoint iterator. The iterator computes a mapping from program points to abstract values. After the fixpoint validator checks it, the value analysis simply computes, for each program point, a numeric interval representing this result, using the `range` function of the numeric domain. The soundness of the value analysis is stated by Theorem 7.

Theorem 7 (Soundness of the value analysis)**.** *Let P be a program, σ be a reachable semantic state, that is, $\sigma \in \texttt{reach}(P)$, and $\texttt{res} = \texttt{value\_analysis}(P)$ be the result of the value analysis. Then, for each program point l, and for each local variable $v \in \texttt{dom}(\sigma.E)$ that contains an integer i (that is, $\sigma.E(v) = \mathit{Vint}(i) \ \vee \ \exists b, \sigma.E(v) = \mathit{Vptr}(b,i)$), this integer is in the interval computed by the analysis: $i \in_{\texttt{Itv}} \texttt{res}[l,v]$.*

[Coq Proof]

The proof of this theorem relies on the properties related to the concretization γ_{Mem} of the abstract values. It uses the result of the fixpoint validator and all the intermediate properties proved for each abstract domain at each of the three layers.

If the value analysis is used as part of another analysis (for instance, in the loop bound estimation described in Section 4.3), then its correctness theorem is an intermediate lemma used to prove the property relevant to the "main" analysis. Informally, this is often interpreted as a variant of "no possible value has been forgotten by the analyzer", or a variant of "a given variable is ensured

not to assume this value at this program point". For instance, in the loop bound estimation, the value analysis ensures the property "at the loop header, these are the only possible values for these variables".

6.3 Evaluation of the Value Analysis

Evaluating the value analysis is a crucial step to ensure it is useful. Indeed, it is straightforward to compute a correct value analysis, simply by returning $\top$ for every variable. Our machine checked theorem does not say anything about the precision of the analysis, not does it say if the fixpoint iteration will terminate. It focuses on the critical property: soundness. Since precision, termination and completeness are not proved in our analysis, they must be evaluated somehow. Applying the analysis to a series of benchmarks is a pragmatic and relevant metric to assess its precision.

In this section, we present the evaluation methodology (goals and comparison criteria) and the results of the value analysis on a set of benchmarks from the CompCert C compiler. The value analysis has been evaluated *per se* (that is, without integrating it as part of another tool) and compared to other state-of-the-art tools performing similar analyses that also compute intervals of machine integers. We start by presenting the goals of this evaluation. Then, we describe the other analyzers used in the comparison and the set of benchmarks used for the evaluation. Finally, we report the quantitative results and explain them.

6.3.1 Evaluation Goals

Our development process for the analysis starts with a simple, imprecise but correct value analysis, and then extends it to improve precision, while maintaining the proofs to ensure correctness is preserved (correctness is mechanically verified by the interactive proof assistant Coq). The main goals of the evaluation process are to ensure the analysis is *precise*, *scalable* and *extensive*.

Concerning precision, we are interested in revealing useful information about the analysis, either directly (e.g. number of false alarms) or indirectly (e.g. size of the inferred intervals for program variables). The ideal (most precise) result is not always known for every program point of interest, so these estimations require some manual inspection. About scalability, we are interested in analyses with industrial applicability, that is, able to handle realistic code bases of at least some thousands of lines of code. The main issue is memory consumption (in critical systems design, static analyses are often performed offline, therefore time constraints are less severe), since requiring more than a few gigabytes of RAM may prevent users from running them. Finally, regarding extensiveness: for embedded systems in particular, features such as dynamic memory allocation, user input, and even recursion may be considered as non-essential, and eliminating them allows for more precise and efficient analyses. Support for features such as floating-point can be added later without directly impacting the analysis.

These three axes (depicted in Figure 6.4) provide different views of the analysis and offer trade-off possibilities. While developing static analyses, as any general software, it is important to have *benchmarks* and *test cases* to evaluate the results. For instance, the Astrée [18] analyzer has been developed with a test methodology in mind. In the case of verified software, where each extra line of code induces several extra lines of proof, testing is even more important to avoid useless code.

Each benchmark and test case is more appropriate to evaluate a given kind of goal. We enumerate here three facets of the evaluation phase that help us attaining the previously cited goals:

1. Micro-benchmarks: small, isolated pieces of code consisting of specific, "tricky" situations that trigger specific limits of the analysis. The code sample about signed and unsigned integer intervals, previously presented in Figure 6.2, is an example of a micro-benchmark.

2. Large benchmarks: based on real code samples, they are necessary to evaluate the scalability of the analysis, and also useful to assess its extent.

3. Configurable features: the use of configuration files, conditional compilation and other optional features to adjust benchmark parameters.

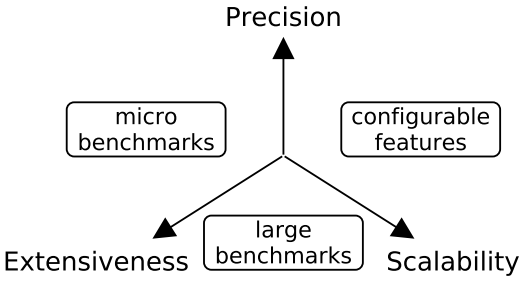


Figure 6.4: The goals for the evaluation of a static analysis can be presented as orthogonal axes. The means to fulfill them, presented inside the rectangles, often involve dealing with multiple axes.

Each of these facets, presented as rectangles in Figure 6.4, relates to two different goals. Micro-benchmarks are useful to test the precision and, in some cases, the extent of the coverage of the analysis. They are also useful as regression tests. However, it is tempting to focus on specific cases which rarely (if ever) arrive in practice, so they should ideally be obtained as program slices of larger benchmarks, i.e. after identifying a real issue in a realistic code base, the issue is isolated as a micro-benchmark.

Large benchmarks, like those developed by SPEC[4], are useful for some analyses, but they are not focused on critical systems, so their code base is not always relevant for this domain. Another limitation of large benchmarks is that they often incorporate several features, requiring an extensive analysis. Some features must either be removed away (for instance, code simplifications related to architecture-specific details), or be made conditionally enabled via configuration settings. Configurable features help to manage extensive benchmarks. These three aspects complement each other for a thorough evaluation of the analysis.

6.3.2 Comparing Different Analyzers

Our comparison between tools serves to identify specific weaknesses (to remediate them if needed), provide inspiration for working solutions (since the tools are open source), and to identify correctness issues in non-verified analyses. We present in this section a case study performed while comparing the following three analyses:

- VVA, Verasco's value analyzer;
- the value analysis performed by Frama-C;
- the value analysis performed by Navas et al. [54], on the SSA form provided by LLVM.

Each of them deals with a different representation for the same C source code. The program variables are different in each of them, but can be related back to the variables in the source. The case study identified some general precautions which must be undertaken for such a comparison. It also suggested some techniques and criteria to help devise similar comparisons.

Verasco's Value Analysis Verasco's value analysis, already presented in Section 6.2, has been put to test by comparing its results with those of other tools.

Frama-C's Value Analysis Frama-C [19] is a software analysis platform for C code aimed at industrial code bases. It has been under development for several years and has been used in different contexts related to static analysis and verification. One of its main components is a value analysis based on abstract interpretation, using several kinds of non-relational abstract domains (strided integer intervals, floating-point intervals, small sets, pointer analysis, etc.).

[4]SPEC, the *Standard Performance Evaluation Corporation*, produces benchmarks for high-performance computing containing large C programs, often from open-source code bases.

Frama-C is based on the CIL [55] front end for C99, which performs code normalization before the value analysis. This simplifies processing the source code, but also has negative consequences due to loss of information caused by some transformations. This is discussed later, in the paragraph *syntactic differences*.

Frama-C has a powerful *semantic unrolling* mechanism, which is similar to syntactic loop unrolling but more efficient. Integrated as a configurable option to the value analysis, it improves precision by re-evaluating a same program point with different contexts without merging them all. Note that some manual tuning of this parameter is required for each program, otherwise setting a default value for all programs risks slowing down the analysis significantly. Since the other analyzers do not have this feature, it has not been considered in the comparison.

Signedness-agnostic Range Analysis ("Wrapped" Tool) Navas et al. [54] developed an interval-based value analysis for machine integers taking into account a combination of both signed and unsigned representations of these integers to improve precision. They use the term *wrapped intervals* for the kind of intervals they deal with. We will refer to their analysis as "Wrapped".

Wrapped's analysis is implemented in LLVM, operating on its intermediate representation. It uses several of LLVM's optimizations, such as constant propagation and function inlining, before the code is sent to the analyzer. This analysis is able to compute precise intervals even in the presence of (deliberate or accidental) overflows.

The Wrapped tool is currently a research prototype which can be used to obtain bounds for variables in a C program. The tool obtains values for the variables it sees on LLVM's intermediate representation, and then we map these variables back to the C source variables.

Source Code Instrumentation Among the three tools we have chosen to evaluate, one of them (Frama-C) operates on the source level (after some preprocessing), while the others operate on specific intermediate representations. Since additional variables are created and source variables may be optimized away, we instrument the source code with analyzer-specific code (in a header file to be included by all benchmarks) and a generic macro `DUMP(str, var)` which must be placed at the program points where we want to examine the result of the analysis. It tracks the interval computed for variable `var` and associates it to the string `str`. For instance, Figure 6.5 presents two code fragments, one from the header file containing the instrumentation code (`bench.h`) and another from a fictitious test case (`test.c`) where we add a call to the `DUMP(str, var)` macro. Inserting a call to `DUMP` does not change the result of the analysis, but adds information to the analysis trace that is later used to retrieve the intervals. In the presence of optimizations, it is sometimes hard to ensure that `DUMP` statements are not removed, since they are semantically irrelevant. For instance, in Figure 6.5, the Wrapped version of the instrumentation uses some tricks (e.g. replacing `s` with `s + 0`) to defeat simple optimizations. More powerful optimizations have to be disabled, which might have a negative impact on the performance of the analysis.

Criteria A very simple criterion for the comparison, which is the length of the inferred intervals for each program variable, produces results which are skewed toward the extremes: most intervals are either bounded to small lengths, or practically unbounded. A *practically unbounded* interval is an interval that is either unbounded or that contains too many elements to provide useful information (e.g. more than 2^{31} elements for a 32-bit integer variable). The former happens because most program variables have a reasonably small variation interval (e.g. loop induction variables, array indices, etc.), while the latter happens when the variable has not been bounded, or has only some boundary conditions which make it *quasi-unbounded*. In practical terms, no useful information has been discovered (e.g. the inferred interval is $[-2^{31}+1, 2^{31}-2]$), but the interval is different from *top*. Due to numerical comparisons, the analysis infers some information about extreme values, but nothing that is practically useful. Figure 6.6 illustrates this situation with a series of nested comparisons refining information about the program variables, but without providing useful information.

The criterion we have chosen to compare analyses is the *number of bounded variables*. It avoids issues with lengths while still providing useful information. This comes from the observation that, in general, when one analysis finds useful bounds for a variable, other analyses either obtain similar

```
// bench.h (analyzer−specific code for instrumentation)
#if defined(__clang__) // for Wrapped
# define DUMP(s, n) { int s; if (s+0) { s = n+0; }} // code to defeat optimizations

#elif defined(__COMPCERT__) // for VVA
void DUMP(const char *s, int n) { int i = n; } // call a useless function

#else // for Frama−C
# define DUMP(s, n) Frama_C_show_each_## s(n) // use Frama−C's built−in

#endif
```

```
// test.c − usage example
#include "bench.h"
void main() {
    int i;
    while (i < 5) {
        i++;
        DUMP("loop_in_i", i);
    }
    DUMP("loop_out_i", i);
}
```

Figure 6.5: Example of source code instrumentation for comparison between analyzers. The first part of the instrumentation is defined once for each analyzer, in a separate header file (`bench.h`). Each source file (here, `test.c`) includes this header and adds calls to `DUMP`.

(if not identical) results, or they fail to bound the variable at all. Also, in the kind of analysis we consider here, failing to obtain a precise result quickly degenerates into $\top$. Therefore, tracking precisely the size of the interval of each variable does not add much useful information.

A variable is considered bounded if its interval contains no more than 2^{31} elements, that is, at most half of all possible integer values. This has been arbitrarily defined to exclude as many useless intervals as possible, without excluding possibly useful ones. Examples of possibly useful intervals are those concerning memory alignment (e.g. $[-2^{31}, 2^{31}] \bmod 4$) and sign information (e.g. $[0, 2^{31}]$). This criterion does not eliminate the need to inspect edge cases (such as e.g. $x \in [-1, 2^{31} - 1]$, considered unbounded but possibly useful), but it greatly minimizes their number.

Benchmark Considerations

The main quantitative evaluation of the value analysis is based on two test suites, extracted from the CompCert benchmarks and from the Mälardalen WCET benchmarks. Each suite consists of 20 C programs. The CompCert benchmarks are divided in two sets: the first one contains 17 single-file programs ranging from 29 to 1,035 lines of code each (totaling about 3,200 lines of code), while the second one contains three multi-file programs totaling slightly more than 2,200 lines of code. The benchmarks we used from the Mälardalen suite are single-file programs ranging from 30 to 2,360 lines of code (totaling about 5,500 lines of code). Finally, we also ran some experiments on a larger CompCert benchmark containing over 3,000 lines of C and about 10,000 Cfg instructions after inlining.

For the evaluation, these programs have been modified in two ways:

- recursive calls have been removed (5 among the 20 CompCert benchmarks contained recursive calls), modifying the semantics of these programs;
- annotations have been inserted at distinctive program points, to track where the inferred variable intervals should be compared between the analyses. Other than generating inspection points, this does not change the semantics of the program.

```
int x, y, z;
if (x < y) {
    // x ∈ [−2^31, 2^31 − 2] y ∈ [−2^31 + 1, 2^31 − 1]
    if (y < z) {
        // y ∈ [−2^31 + 1, 2^31 − 2]
        if (x < y) {
            // x ∈ [−2^31, 2^31 − 3]
        }
    }
}
```

Figure 6.6: Code example illustrating *quasi-unbounded* intervals. After each comparison, the intervals are narrowed by one unit. In practice this information is useless, so these intervals should not be considered bounded.

Since CompCert benchmarks are not oriented for embedded devices, there is a significant number of programs with recursive calls. Neither Frama-C nor VVA currently support recursion in a sound way, and Wrapped does not compute precise bounds in the presence of recursion. As long as the modified program semantics is still defined, this modification should not visibly impact the relative performance of the analyzers.

For the comparison, we defined four kinds of *points of interest*, each of them associated to a subset of the integer variables in the program:

1. function entry, associated to integer arguments
   (e.g. for a function with signature `void f(int a, int* b, char c)`,
   we consider the values of a and c when entering the function);
2. function return, associated to integer return values (if any);
3. loop entries, associated to integer induction variables[5];
4. loop exits, associated to integer induction variables.

Other points of interest might include conditional branch entries and *merge points* (where conditional branches merge their results), but inserting annotations at these points requires the ability to easily identify them in the source code, and also a decision about which variables to consider: only those used in the branch condition, or other variables as well? The main reason for not including these points of interest is that they have a worse benefit-cost ratio than the previous ones: it is harder to instrument them, there are many more uninteresting branch/merge points than loops, and most interesting variables are already covered by one of the four main criteria.

6.3.3 Experimental Results

After instrumenting the code, we ran each analyzer, parsed its output, extracted the relevant intervals and computed the number of bounded/unbounded variables per program. We obtained two kinds of data: numerical results (number of bounded intervals) and qualitative differences in behavior between the analyzed tools. These differences entail quantitative disparities between the tools and need to be taken into account. We detail these differences here, both to illustrate some inherent difficulties of the comparison of different tools, and also to justify in part the numerical results.

Semantic Differences Although all three analyzers are based on C code as input, they have significant semantic differences. As an example, we cite the behavior of the Frama-C value analysis. In Frama-C, possibly undefined behaviors are dealt with in two steps:

[5]Pointer-based loops, such as the traversal of a linked list, are not considered.

1. when the analysis arrives at a point where undefined behavior is possible, Frama-C emits a *warning*, notifying the user about the possibility of undefined behavior. For instance, if the expression `x << y` is encountered, with $y \in \top$, Frama-C emits a warning stating *invalid RHS operand for shift. assert* $0 \leq y < 32$. Note that, in C, the bitwise shift operators require that the second operand (the number of bits to be shifted) must be between 0 and the bit-width of the type of the variable being shifted (e.g. 8 for a `char`, 32 for a `float`);
2. the analysis inserts an *assumption* that filters out all undefined behaviors. From that point on, the analysis considers only a *subset* of the behaviors of the original program. In the previous example, Frama-C assumes $y \in [0, 31]$ at the program point originating the warning, and propagates this information.

This allows Frama-C's value analysis to strengthen its bounds on some variables, since it has now the additional hypothesis that *no undefined behaviors will be triggered by the statement where the warning has been emitted*. While this behavior is consistent with the assumptions of most modern compilers, which exploit such situations to more aggressively optimize the compiled code, this is not the approach taken by the other analyzers. This difference complicates direct comparisons between Frama-C and the others.

Other semantic differences between the tools include behavior with respect to signed integer overflows: Wrapped is designed to accept and handle such overflows using two's complement arithmetic, while Frama-C follows the ISO C standard, which states that such behavior is undefined. Finally, another source of semantic differences is the usage of standard library functions. In VVA, some standard C functions such as `malloc` and `memcpy` have their semantics specified by the CompCert compiler. Our analysis uses this semantics to infer some properties about these functions. Frama-C has a set of standard library stubs which contain ACSL[6] annotations describing their behavior. Frama-C can therefore obtain some extra information, for instance by knowing that the return value of the `abs` (absolute value) function is a non-negative integer. Wrapped does not have any stubs or built-ins to obtain such information. All in all, these semantic differences serve as a reminder that a direct comparison between tools is subject to several aspects which are not immediately obvious and require careful consideration.

Syntactic Differences Besides semantic difference, we also identified a difference due to a syntactic modification of the code that takes place before the value analysis. We describe it here as an illustration of a semantics-preserving modification that has a measurable impact on the performance of the analysis.

Frama-C's front end (based on CIL) includes preprocessing for the C logical operators `&&` (*and*) and `||` (*or*) in conditional expressions, which works as in the example below, where the code in the left column has been transformed into the code in the right column.

```
if (a && b || c) {
    r = 1;
}
```

```
if (a) {
    if (b) { goto _LOR; }
    else { goto _LAND; }
} else {
    _LAND: /* internal */ ;
    if (c) { _LOR: /* internal */ r = 1; }
}
```

This kind of transformation for short-circuiting conditional operators is commonplace (for instance, in CompCert it is performed during back-end compilation), but it can have a negative impact on the analysis: each conditional only deals with one variable, therefore any relational information that could have been obtained from the `a && b || c` expression is lost. Because Frama-C uses non-relational abstract domains (intervals), this information is not recovered by the analysis. Either removing the transformation or using a relational value analysis (e.g. polyhedral domains) would provide a solution to this issue. In our evaluation, we found some code samples (e.g. `knucleotide`) where this prevented Frama-C from getting the same results as VVA, which incorporates the *numeric expression trees* mentioned in Section 6.2.3, without splitting.

[6] ANSI/ISO C Specification Language [7]. It is an annotation language for C, based on a *design-by-contract* model. ACSL is used to specify properties for verification tools based on program proof.

Quantitative Results The numerical results of the evaluation, in number of bounded intervals per program and per analyzer, are displayed in Figure 6.7 for the CompCert benchmarks and in Figure 6.8 for the Mälardalen WCET benchmarks. In total, there were 866 points of interest in the considered programs, 545 in the CompCert benchmarks and 321 in the Mälardalen benchmarks. Among those, Frama-C bounded 695 variables, VVA bounded 591 variables and Wrapped ended up with 538 bounded variables.

The main differences between VVA and Frama-C, especially on the larger benchmarks (`lzw`, `arcode` and `lzss`), come from global variable tracking and congruence information. These are handled by Frama-C, but not by VVA. On `fannkuch`, the difference between Frama-C and the other analyzers is due to its widening operator, which is configured for fast convergence. This is alleviated by providing *hints* (via manual annotations in the source code) or by using the *semantic unrolling* option which we did not enable. Also in this program (and in some others), Frama-C bounded fewer variables due to the preprocessing of conditional operators, as previously discussed. Some issues with the inlining used by Wrapped explain its worse results in `fft`, `knucleotide` and `spectral`. In fact, Wrapped has not been optimized to deal with such an aggressive inlining. Also, some LLVM optimizations were disabled to allow us to obtain useful information from the annotations. Despite local tests indicating that there was no loss of precision for the kind of programs present in the benchmark, we cannot exclude the possibility of such loss happening in larger benchmarks. Unfortunately, it is unfeasible to manually compare the optimized LLVM code with the original one in this case.

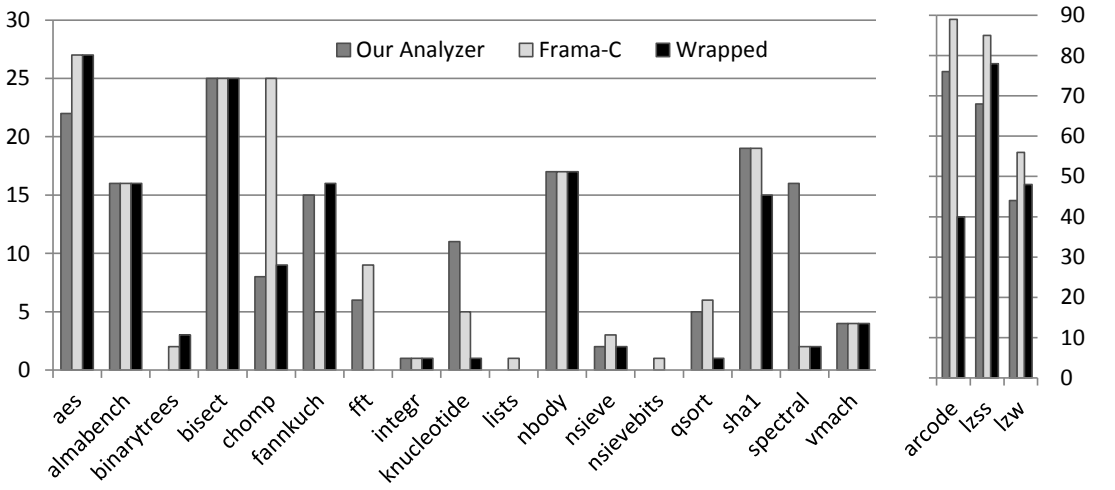


Figure 6.7: Number of variables considered bounded by the value analyses on the CompCert benchmarks, per test program and per analyzer.

With respect to execution times, we observe that VVA is faster than Frama-C's for programs containing several pointers and global variables, because VVA does not track this information while Frama-C does. For other programs, both analyses run in roughly the same time: less than a second (about 0,1 – 1s) for the smaller programs and a few seconds (1 – 20s) for the larger ones. Wrapped's analysis is consistently faster than the others, but in some programs the aggressive inlining can generate an explosion in memory usage. Finally, tests performed on a larger benchmark (containing over 3,000 lines of C code and about 10,000 Cfg instructions after inlining) took 34 seconds for VVA.

6.3.4 Continuing Work on the Value Analysis

A new version of Verasco's Value Analysis is under development for the Verasco project. This version includes additional memory information and more sophisticated domains, such as strided and floating-point intervals. It also operates on Clight, which is closer to C than Cfg. The development methodology presented in this chapter is being used to help evolve the analyzer. Large benchmarks, such as the *debie1* benchmark developed for WCET estimation tasks, are being modified to integrate

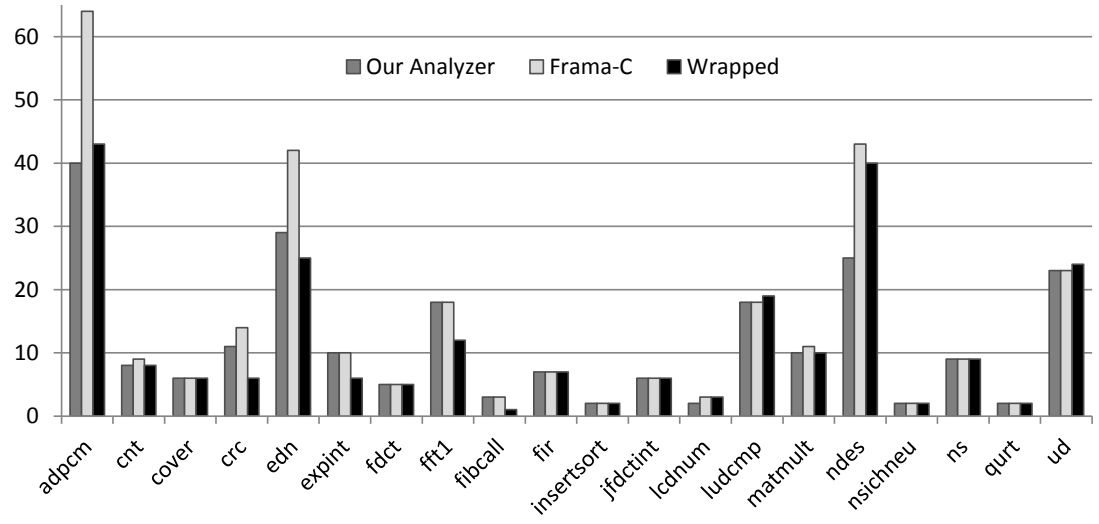


Figure 6.8: Number of variables considered bounded by the value analyses on the Mälardalen benchmarks, per test program and per analyzer.

configurable features such as conditional compilation of parts of the program and emulation of unimplemented operators. We use CompCert's reference interpreter to see if each variant of the program still has well-defined behaviors. Afterwards, we intend to pursue the value analysis with another benchmark, the code of the PHEBUS ultraviolet spectrometer [14], which is similar to *debie1* in several aspects (both originate from space-related scientific software) but contains extra challenges due to its "raw" nature (i.e. it has not been simplified to constitute a benchmark).

6.4 Conclusion

A value analysis is a powerful and versatile tool for static analysis, not only for WCET estimation (where it has several uses [71]), but also to establish properties such as memory safety. Proving correct a value analysis for a language as expressive as C is a complex task, which can be greatly simplified by applying clever techniques such as: concentrating on the primary property, which is correctness, without proving others (precision, efficiency and termination can be experimentally evaluated without having to be proved); applying *a posteriori* validation when it is worthwhile; and performing proof and evaluation in parallel, using experimental results to guide its evolution. The modular architecture of the Verasco analyzer presented here enables the replacement of abstract domains with more sophisticated ones, and the use of domain products allows the integration of specialized domains to compensate for eventual weaknesses in other domains. We establish the general correctness theorem of the value analysis, which will then be used by the other analyses (such as the loop bound estimation) to prove their correctness. In terms of precision, to ensure the analysis performs non-trivial reasoning, we performed an experimental evaluation by comparing it to other tools computing similar analyses. Due to the differences between tools, our evaluation required the consideration of several aspects, such as semantic and syntactic differences, which prevent a direct comparison of the precision of each tool. Still, we were able to establish some criteria which allowed us to conclude that our analysis performs non-trivial reasoning, and future evolutions will be able to attain more precision, including notably floating-point and memory domains.

Chapter 7

WCET Estimation, Implementation and Evaluation

In this chapter, we concentrate on the final part of the WCET estimation tool, the IPET-based WCET estimation. We also detail the implementation effort in this thesis and present some experimental evaluation related to both loop bounds and WCET estimation.

In Chapter 3, we presented the general architecture of a WCET estimation tool, divided into three parts: control flow analysis, processor-behavior analysis and bounds estimation. The control flow analysis has been dealt with by our loop bound estimation, formalized in Chapter 4. Its results—formally verified loop bounds at the assembly level—are now used by our IPET-based WCET estimation. In Section 7.1, we present a pen-and-paper formalization of this technique. Then, in Section 7.2, we focus on the implementations developed in this thesis and on their integration with CompCert. Finally, in Section 7.3, we present the evaluation of our loop bound estimation on the Mälardalen WCET benchmarks, comparing our results with those obtained by SWEET. We also present some experiments related to the precision of our WCET estimation.

7.1 Estimating the WCET

The WCET estimation is performed at the CompCert language closest to the hardware, that is, assembly. Our current implementation uses the PowerPC assembly language and semantics, but it can easily be adapted for the x86 or ARM variants of CompCert. The diagram in Figure 7.1 presents an overview of the estimation, whose general architecture has already been described in Section 3.1.3. Our Cfg program is compiled to assembly, along with the estimated loop bounds. Timing costs are provided by an external hardware model. All this information is converted into linear inequalities—constraints between program points and timing costs—forming an ILP system. These inequalities are then sent to a solver which searches for the maximal solution and outputs it as the WCET estimation.

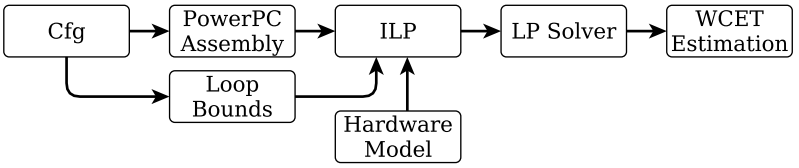


Figure 7.1: Overview of our WCET estimation via an ILP. The Cfg program with its loop bounds is compiled to the PowerPC assembly; combined with a hardware model, they are used to create an ILP system whose maximal solution, given by an LP solver, is our WCET estimation.

Our method is based on a simple hardware model, described in Section 7.1.1. The formalization of the constraint generation for the ILP system, as well as an efficient validation technique to ensure its correctness, are described in Section 7.1.2.

7.1.1 Hardware Model

As presented in the context (Section 3.1), formalizing a hardware model with cycle-accurate timing information is a complex task which is feasible for simple architectures, such as some 8-bit microprocessors (e.g. Intel 8051, which has neither cache nor pipeline), but out of reach for most current architectures. In particular, more complex configurations often introduce so-called *timing anomalies* [62], situations where what would intuitively be considered as the worst-case scenario (e.g. a cache miss) does not lead to the highest actual execution time. More precisely, the assumption of a *local* worst-case scenario does not necessarily lead to a *global* worst case. This is due to the interdependence between different timing-related elements (e.g. an instruction prefetcher, which in average increases execution speed, but that in some cases leads to cache misses, negatively affecting the execution time). In addition, modern architectures are optimized for the *average* case, even if it implies degrading the worst case, or making it less predictable. These facts complicate the development of hardware timing models.

Due to the specificity of timing models (they closely depend on the hardware configuration, therefore they cannot be easily transposed to other architectures), WCET estimation tools often accept several hardware configurations, which are easily switchable. Some analyses are parameterizable, either to allow for architecture variants (with different cache sizes, for instance), or for development purposes (e.g. debugging). Our processor-behavior analysis fits into this model: by default, we adopt a simple cost model, where each instruction is associated to a single clock cycle. The estimated WCET is therefore equivalent to the number of executed instructions. Formalizing these models, including the the semantics of cycle-level timing information, is left open for future work and integration.

7.1.2 IPET Formalization

Our application of IPET to produce a WCET estimation is based on the following steps:

1. instrumentation of the assembly language semantics;
2. incorporation of a hardware timing model;
3. generation of constraints associated to execution counters;
4. definition of the objective function estimating the WCET;
5. solution of the objective function via a verified method.

We describe each of these items along with some aspects related to their formalization.

Instrumentation of the Assembly Language Semantics

The assembly language semantics (in the current implementation, PowerPC) is instrumented by adding execution counters for each program point and program transition. These counters start at zero and are incremented whenever a given program point is reached, or a program transition is taken. They are equivalent to the global bounds of the instrumented ICfg language, except that in ICfg only counters for program points were considered, while here we also include counters for transitions. More precisely, these counters are associated with an implicit control flow graph of the assembly program, where transitions correspond to the edges of the CFG, and program points to its vertices. Note that our CFG is computed statically. `switch` statements are compiled using *binary decision trees*, which avoids the dynamic jumps of jump tables. Instructions using dynamic jumps[1] are not dealt with.

The following step is the incorporation of a hardware model. As stated previously, our model is simple, with a unitary cycle cost, but it does not preclude the choice of a more realistic hardware model in the future. The hardware model is chosen before the generation of the constraint system, because its choice may lead to the creation of additional constraints, as we explain next.

[1]Such instructions are typically produced when using function pointers. They do not occur, for instance, in the Mälardalen benchmarks.

ILP Constraints

To produce a static approximation of the execution counters, we define a variable x_i for each program point i and a variable $e_{i,j}$ for each program transition from i to j. They approximate their respective execution counters. With these variables, we generate integer linear constraints approximating all possible execution flows of the program. This approximation is *safe*, that is, it is an over-approximation of the actual execution flows, which ensures the estimated WCET will be an upper bound of its actual value.

Structural constraints, relating program points to their CFG predecessors and successors, have the form $\sum_{p \in preds(i)} e_{p,i} = x_i = \sum_{s \in succs(i)} e_{i,s}$. They are obtained directly from the CFG of the assembly program. Semantic constraints, of which there is at least one per loop in the program, have the form $x_i \leq N$, where N is the loop bound. The values used are those provided by our loop bound estimation.

To prove that these constraints are valid, we need to relate them to the instrumented semantics. For instance, showing that a given constraint $x_i \leq N$ is valid means proving that no program execution can reach x_i more than N times. Other semantic constraints can be added to improve precision, such as constraints related to conditional branches, or relational loop constraints of the form $x_i \leq x_j * N$, where j is in a loop nested within the loop of i. Other useful constraints are related to the hardware timing model.

Constraints from the Hardware Model In our simplified timing model, we estimate the number of executed instructions, which is obtained directly from the variables x_i associated to the execution counters. However, the IPET approach is generic and accepts more complex hardware timing models with few modifications.

In our simple model, we assume $t_i = 1$ for every program point i. This is sufficient to evaluate the result of the IPET and the WCET estimation. Note that this solution, while more precise than counting one cycle per basic block, is also less scalable. Alternatively, we can associate the number of instructions in a basic block to a timing constraint, and then count only basic blocks.

Correctness Proof

We present a pen-and-paper formalization of the properties necessary for the correctness proof of the ILP constraints. With respect to the structural constraints, the following property holds for every assembly program P:

$$\forall i \in \textit{inner-vertices(P)}, \sum_{p \in preds(i)} e_{p,i} = x_i \leq \sum_{s \in succs(i)} e_{i,s}$$

inner-vertices is a function which returns all vertices in the program, except the entry and exit vertices, which have special constraints: $x_{\text{entry}} = 1$ and $x_{\text{exit}} = 1$. The first one states that program execution must begin at the entry point; it also states that this entry point does not have any predecessors, that is, it is outside of any loops. This is guaranteed via CFG normalization at the Cfg level and it can be validated at the assembly level. The constraint on the exit node is related to the fact that the program is supposed to terminate. *preds* and *succs* return the predecessors and successors of a program point, respectively. They are calculated by the CFG reconstruction. Note that there is an inequality between x_i and $\sum_{s \in succs(i)} e_{i,s}$. This property is weaker than the usually stated equality between these elements. This is useful for the inductive proof based on the execution trace, and it does not affect the final result. The proof by induction consists in showing that, whenever execution advances one step, the inequality holds for every program point. Since the only program point for which it could no longer hold is i itself, since x_i has been incremented, the fact that the variable associated to edge $e_{i,s}$ is also incremented preserves the invariant.

The proof related to the semantic constraints is a consequence of the main theorem of the loop bound estimation, Theorem 2 presented in Section 4.4. Our proof consists in showing that the negation of any semantic constraint would imply non-termination of the program. Note that this proof is related to the program point associated to the constraint, independently of the loop where it is located. In particular, we do not need to prove that every loop has an associated semantic

constraint (this would require proving the correctness of the reconstruction of the loop structure at the assembly level). As expected, this proof does not show completeness (a program where a loop has not been bounded will produce an infinite WCET), but it avoids unnecessary proof effort.

Objective Function

The conjunction of all ILP constraints produces a system whose solutions represent different program execution flows, or more precisely, static overapproximations of such flows. To obtain the estimation of the program execution time, we simply need to add the estimation of the execution time of each individual program point (t_i) multiplied by the (estimated) number of times it is executed (x_i): $\sum_{i \in \text{vertices}} x_i.t_i$.

The WCET is the execution time of the *worst* possible execution. Its estimation is therefore the *maximal* solution for the timing function. Therefore, we consider it as the objective function of a linear program which we seek to maximize.

We need an LP solver to compute our WCET estimation, but we do not need to formalize the solver itself; such task is out of the scope of this thesis. Instead, we use an approach based on *Farkas' certificates* [8]. This approach ensures that a solution is maximal by demonstrating the absence of larger solutions.

Farkas' Certificates A Farkas' certificate is a certificate of infeasibility of a linear program. It is based on the Farkas' lemma, which is presented below. It is applied to a linear system $Ax = b$, of m inequalities and n variables, where $A \in \mathbb{Q}^{m \times n}$ is the coefficient matrix, x the vector of unknowns and b the vector of constant terms. Farkas' lemma does not help us find a solution to the system; instead, it states that, if we are able to obtain a certificate $c \in \mathbb{Q}^m$ (a column vector) which respects a few properties, then the system is infeasible. More specifically, the lemma states that (1) given a certificate c such that (a) every coefficient in c is non-negative, (b) the product of A^t and c is zero, and (c) the scalar product of b^t and c is strictly positive, then we can assert that (2) there exists no feasible solution to the system ($Ax \neq b$ is a consequence of the lemma).

Lemma 5 (Farkas' Lemma)**.** *Let $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. The following statements are equivalent:*

1. $\exists c \in \mathbb{Q}^{+^m}$ *s.t.* $A^t c = \bar{0}$ *and* $b^t c > 0$.
2. $\forall x \in \mathbb{Q}^n, \neg(Ax \geq b)$.

Farkas' lemma establishes the equivalence in both directions, but for correctness we only need to prove that (1) $\Rightarrow$ (2), that is, given a certificate, we show how this certificate ensures infeasibility of the system. The other direction ((1) $\Leftarrow$ (2)), which is the hardest part to verify, is only necessary for completeness of the method and not needed in our lightweight proof approach.

The certificate is a set of constant terms which, when multiplied with the coefficient matrix, reduces it to a state of trivial infeasibility. More precisely, we can write $Ax = b$ as $x^t A^t = b^t$, then multiply both sides by the certificate, obtaining $x^t A^t c = b^t c$. Conditions (b) and (c) ($A^t c = \bar{0}$ and $b^t c > 0$) ensure that we arrive at a contradiction. Therefore, $Ax = b$ has no solution. This approach is based on the result certification for polyhedral analysis by Besson et al. [8] and it provides an efficient way (in terms of proof effort) to verify the result of an untrusted external solver. The extra necessary computations are the certificate generation plus its verification, which comprises:

- computation of the matrix-vector product $A^t c$;
- verification that the resulting vector is a null vector;
- computation of the scalar product $b^t c$;
- verification that the result is strictly positive.

From the certificate, one can conclude the infeasibility of a linear system. But what we actually want to prove is that a given solution of the objective function is maximal. Therefore, we add the *negation* of the solution to the linear system, and then we prove that the new system is infeasible. More precisely, we proceed in the following manner:

1. compute, using an external LP solver, the maximal solution to the objective function; this is the WCET of the program, which is untrusted;

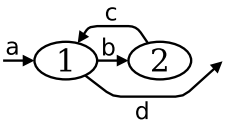
2. augment the linear system with the negation of the solution, e.g. if the estimated WCET formula is $\sum x_i \leq T$, add the inequality $\sum x_i > T$ to the system;

3. compute an infeasibility certificate for the augmented system (which is another linear programming problem);

4. give the certificate to a verified validator based on Farkas' lemma, which uses it to show that the augmented system is infeasible;

5. conclude that the result obtained by the non-augmented system is a maximal solution, and therefore a correct WCET estimation.

Example We present an example of how the certificate works. Let us consider a very simple program, consisting of two program points and a loop (which we suppose has already been bounded to 5 iterations):



To keep our example small, we will only use the edges as variables for the ILP, and we do not include the hardware model. Adding variables for program points or hardware cost coefficients does not alter the method.

The ILP constraints generated for the CFG edges are: the initial constraint $a = 1$; the structural constraints $a + c = b + d$ (program point 1) and $b = c$ (program point 2), and the loop bound constraint $c \leq 5$. The function to maximize in this case is $a+b+c+d$. We suppose these constraints and the objective function have been given to an external LP solver, which output the solution $(a, b, c, d) = (1, 5, 5, 1)$, for a WCET of 12. We wish to verify that it is indeed a maximal solution.

To apply Farkas' lemma, the first step is to normalize the constraints into inequalities of the form $x \geq y$ (splitting $x = y$ into $x \geq y$ and $y \geq x$, for instance). The resulting system contains therefore 8 equations, and in matrix form is written as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \geq \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -5 \\ 13 \end{pmatrix}$$

For instance, the fifth and sixth rows correspond to $b - d \geq 0$ and $-b + d \geq 0$ respectively, which are derived from the constraint $b = d$. The seventh row ($-c \geq -5$) comes from the loop bound constraint multiplied by -1, and the last row is the negation of the WCET formula.

We suppose a given certificate k has been generated by an external solver. In practice, the same ILP solver used to compute the solution can also produce such certificates, by solving $A'^t k = \bar{0}$, where A' is the matrix A normalized and augmented with the negation of the WCET solution. Note that certificates are not unique. For instance, a suitable certificate in this case is $k^t = \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 2 & 1 \end{bmatrix}$. If we multiply the equations by these coefficients, we obtain:

$$
\begin{array}{lrrrrcr}
(0\times) & a & & & & \geq & 1 \\
(2\times) & -a & & & & \geq & -1 \\
(1\times) & a & -b & c & -d & \geq & 0 \\
(0\times) & -a & b & -c & d & \geq & 0 \\
(0\times) & & b & & -d & \geq & 0 \\
(0\times) & & -b & & d & \geq & 0 \\
(2\times) & & & -c & & \geq & -5 \\
(1\times) & a & b & c & d & \geq & 13
\end{array}
\quad\Longrightarrow\quad
\begin{array}{rrrrcr}
 & & & 0 & \geq & 0 \\
-2a & & & & \geq & -2 \\
a & -b & c & -d & \geq & 0 \\
 & & & 0 & \geq & 0 \\
 & & & 0 & \geq & 0 \\
 & & & 0 & \geq & 0 \\
 & & -2c & & \geq & -10 \\
a & b & c & d & \geq & 13 \\
\hline
 & & & 0 & \geq & 1
\end{array}
$$

And thus we arrive at a contradiction, which confirms that the original WCET estimation ($a + b + c + d \leq 12$) was maximal. The final correctness theorem of WCET estimation is presented below.

Theorem 8 (Correctness of the WCET estimation)**.**
Let P_{Asm} be an assembly program, and C the set of structural and semantic ILP constraints related to the program. Then, for any finite execution of P_{Asm}, considering the timing cost function T and the static approximation X of the execution counters of P_{Asm},

$$\max \sum_{i \in vertices(P_{Asm})} X(i).T(i)$$

is a correct over-approximation of the execution time of all terminating executions of P_{Asm}.

We do not formally prove the precision nor the completeness of the WCET estimation. To ensure our method produces useful results in practice, we performed an experimental evaluation on the same set of benchmarks used for the evaluation of the loop bound estimation. This evaluation is presented in Section 7.3.2.

7.2 Implementation

The implementation, testing and experimental evaluation of our analyses (mainly WCET estimation and loop bound estimation) constitutes a significant development in this thesis. Integrating all of them into CompCert culminated in a tool able to perform a WCET estimation, plus several program transformations and static analyses (mainly loop inversion, program slicing and value analysis). This tool also provides extra features, such as computing the execution trace of the compiled assembly program. In this section, we present the tool as an extension of CompCert, with the associated options and analyses it offers; then, we detail some implementation aspects related to its integration.

7.2.1 Overview

The CompCert compiler is run by executing a driver program called `ccomp`. `ccomp` is responsible for processing command-line options (e.g. setting input and output options and filenames), invoking the C code parser, and then calling the compilation function, which has been generated from the Coq proved development. This compilation function processes the AST produced by the parser and invokes each compilation pass in turn, until the final assembly code is produced. We integrated into `ccomp` our loop bound estimation, as an optional compilation pass which does not modify the final program but produces an extra information: execution bounds associated to the assembly program.

The loop bound estimation is activated by specifying an extra command-line option (`-bounds`) when running `ccomp`. Since it does not modify the code, it does not affect the correctness of the entire compilation chain. Before running the loop bound estimation, the user inserts annotations in the source program, placing them in the program points for which he or she wants to obtain bounds, typically at each loop. The result of the loop bound estimation is a mapping from these annotations to the estimated bounds. For instance, consider the following program:

```
int i = 0;
do {
  int j = 4;
  _annot("loop1");
  do {
    _annot("loop2");
    j--;
  } while (j > 0);
  i++;
} while (i < 5);
```

Running this program with `ccomp -bounds` will produce the following output: `loop1 -> 5` and `loop2 -> 20`. These are the estimated global bounds for the program points containing the annotations, which correspond to the loops in the program.

Besides global loop bounds, there are several interesting analyses and program transformations which are provided by our tool. It can perform loop transformations (inversion and unrolling), reconstruct the loop structure of a Cfg program, compute a program slice, perform a value analysis, display *local* loop bounds, interpret an assembly program and display its execution trace, produce ILP constraints for an assembly program, and compute its WCET estimation. These features operate at different levels in the CompCert compilation chain, involving several intermediate languages. Figure 7.2 displays an overview of the location of each analysis and transformation in the compilation chain.

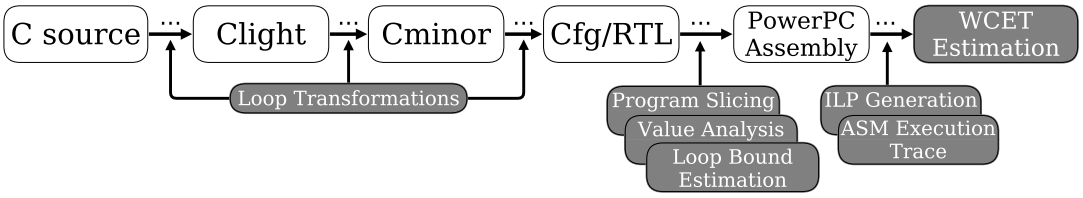


Figure 7.2: Overview of the CompCert compilation chain, with the languages relevant to this thesis and the integration points of our (optional) passes.

Except for the verified loop bound estimation, all other analyses have been implemented in a separate driver program, `cfgcomp`, which is very similar to `ccomp` but contains different command-line options. Separating them from `ccomp` is useful during development, to avoid tainting the proved parts with development code. Another reason for the separation is to maintain the conciseness of the correctness theorems of the elements in `ccomp`. For instance, if `ccomp` provided local bounds estimation, the main correctness theorem of the loop bound estimation (Theorem 2, in Section 4.4) would include local bounds in its definition, which would make it less clear. This is also the reason why annotations are not directly inserted at each loop header in `ccomp`. By implementing a separate version of the loop bound estimation in `cfgcomp`, with a different processing of the input/output data (but otherwise reusing the same proved code), we obtain a more complete version which displays local bounds.

Figure 7.3 depicts the overall structure of both `ccomp` and `cfgcomp`, distinguishing between existing parts, represented in red rectangles (the compilation chain and external tools such as an LP solver) and the newly-implemented ones, represented in white rectangles. Various command-line options enable the activation of analyses and transformations, several of which can be combined, such as loop transformations and bounds estimation. Some optional passes have prerequisites, i.e. the ILP generation requires loop bounds. Overall, each output indicated in the right side corresponds to an execution mode, which can be considered as a separate analysis. Each of them is presented in more detail in the next section, with practical usage examples.

Development Metrics We have integrated our loop bound estimation in the CompCert 1.11 compiler. Our formal development, including the parts related to the other analyses necessary for loop bound estimation (loop reconstruction, program slicing and value analysis) and WCET

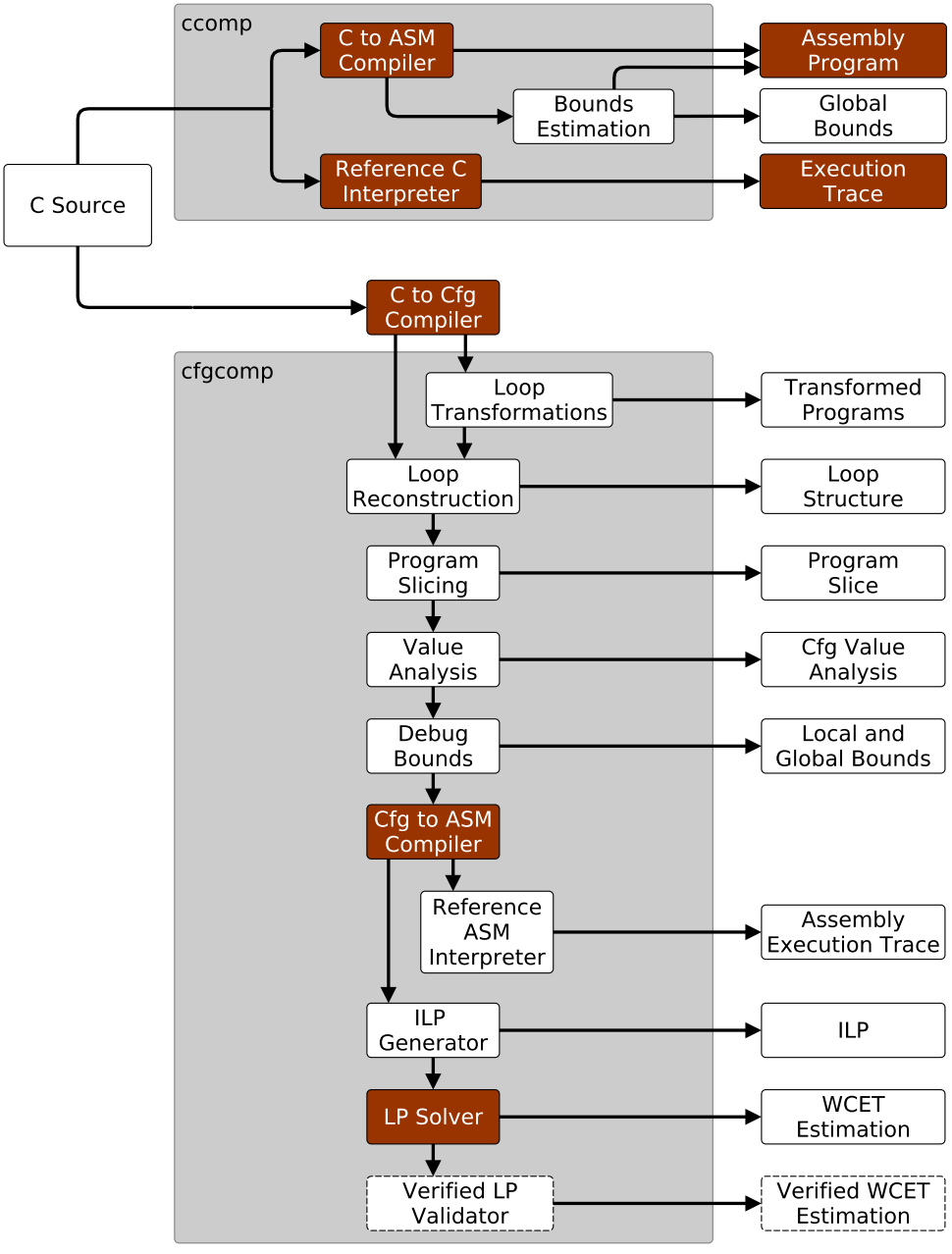


Figure 7.3: General view of the `ccomp` and `cfgcomp` programs, with focus on the optional passes implemented in this thesis. Red rectangles indicate existing components. Dashed rectangles indicate elements not yet finalized.

estimation, consists of about 8,000 lines of Coq functions and definitions, and 12,000 lines of Coq statements and proof scripts. Additionally, there are about 5,000 lines of non-verified OCaml code, including the `cfgcomp` driver, untrusted components whose result is later validated (such as the program slice set builder), and auxiliary functions (such as pretty-printers).

7.2.2 Analyses Performed by cfgcomp

We describe here the main aspects of each analysis performed by `cfgcomp`, with examples where they might be useful.

Loop Transformations The loop transformations are performed during compilation from C to Cfg, and consist in applying *loop inversion* and *loop unrolling* (with a user-defined unrolling factor) to the program to be analyzed. The actual effect of these transformations is described in Section 7.3.2, where they are presented as optional passes to increase the precision of the WCET estimation. However, these transformations can be applied independently of the WCET estimation, for instance to compare their efficiency with respect to optimizations or other analyses.

Loop Reconstruction Loop reconstruction at the Cfg level may have applications for any other static analysis. Information about loop headers, loop exits, branches inside loops, etc., can be derived from the loop structure and applied for optimizations or to derive heuristic information about the structure, e.g. to help estimate which loops might benefit from unrolling, and which ones are too large and would be prohibitively expensive. Even basic information such as loop nesting level can help development; for instance, we used this information to pretty-print the Cfg code with loop nesting information, which is sufficient in most cases to restore the control flow information.

Program Slicing Program slicing has numerous applications, and since it is constituted of several independent steps (such as postdominance tree, control and data dependency computations, and program dependency graph construction), in principle any of these could be offered to the end user, although we did not provide this directly. We do allow the user to compute a slice with respect to a given program point, which may be helpful to identify useless code, for instance, or to help tracking dependencies.

Value Analysis A value analysis by itself, as mentioned in Chapter 6, has several uses. It is also the basis of some analyses, such as the loop bound estimation. For these reasons, it is the default analysis performed by `cfgcomp`. Its result is output as an annotated version of the Cfg program, which displays the variation domains (in the form of intervals) for program variables at each program point.

Local and Global Bounds As explained before, the `cfgcomp` version of the loop bound estimation is more customizable than the `ccomp` version, providing automatic insertion of loop annotations and displaying local bounds. Also, the set of interesting variables, which contribute to the loop bound estimation, can be more easily obtained in this version.

Assembly execution trace The execution trace of the assembly program is especially useful for the WCET estimation. For instance, it allows to obtain the number of instructions executed in the program. A reference interpreter for assembly is available in `cfgcomp`. It is similar to the C interpreter in `ccomp`, but more useful for a WCET estimation based on the assembly code. If a hardware model is integrated into the WCET estimation, then this model can also be used by the interpreter, enabling an exact count of the number of clock cycles during execution of a program. The reference interpreter, via the execution trace, also helps to identify the worst-case path for a given set of input values.

ILP Sometimes, when applying IPET to estimate the WCET, it may be useful to obtain the constraint system itself, instead of just the final output, the WCET estimation itself. Having access to the constraints allows for some tinkering, e.g. the manual addition of extra constraints representing flow facts to see if they provide noticeable gains, before actually implementing and proving their generation. `cfgcomp` provides a means to obtain the constraint system in the format expected the LP solver tool `lp_solve`.

WCET Estimation Currently, the final result of the `cfgcomp` tool is the WCET estimation produced by the `lp_solve` tool, but finalization of the verified WCET validator will allow the production of a verified result, at which point it will be integrated into `ccomp`, leaving the unverified result in `cfgcomp` for rapid prototyping and testing of other improvements.

Overall, `cfgcomp` is a general-purpose static analysis tool with several options and a useful complement to `ccomp`. Its development enables faster prototyping of new analyses and improvements to existing analysis. In a verified development, it is costly to prove each modification, only to later realize that it does not bring the expected benefits in terms of precision or performance. By maintaining two roughly equivalent versions, one proved and another unproved, we can communicate changes in both directions (e.g. while proving, if we notice that some modifications could simplify the proof, we can quickly perform them on the non-proved version and test) and provide more features for the end user.

7.2.3 Integration with CompCert

We describe in this section some general aspects related to the CompCert integration, as they might interest future CompCert users. We also present some unforeseen difficulties and concepts of the CompCert architecture which might help or hinder a development.

Language Choice The first step for a static analysis in CompCert is usually the choice of which language it will operate on. CompCert has many intermediate languages, but in our case the choice was quickly narrowed down to the Cfg language, since it contains an explicit control flow graph. Higher-level languages such as Clight and Cminor contain structured loops, which avoids the need to reconstruct them; however, they do not have explicit program points and are thus less suited for our analyses.

System Calls The semantics of *system calls* in CompCert is handled in a particular way. Functions such as `malloc`, `free`, `memcpy`, etc., which handle the memory in a very precise way (necessitating a special semantic definition), are specified axiomatically, and this specification is inevitably incomplete. This gives rise to the need of extra hypotheses in the proof. As a specific example, we consider the assumption that, in a deterministic environment, calls to `memcpy` with a same memory state will not depend on the event trace. Indeed, it is reasonable to consider that a correct implementation of a semantically well-defined call to `memcpy` (which excludes the use of uninitialized or volatile variables, for instance) will not depend on the execution trace of the events that happened before it, but only on the memory state itself. CompCert does not need this property and therefore does not state it, but if it becomes necessary for some specific analysis, it needs to be assumed for the proof. We added such an assumption to avoid losing precision when dealing with programs containing these system calls: without the assumptions, the analysis conservatively overestimates their behavior and seems to perform poorly when compared with non-verified analyses. The assumption is necessary for a fair comparison between them.

Updating a Verified Development CompCert is constantly evolving: during the course of this thesis, there have been several releases of new CompCert versions, with some of them bringing novelties having an impact in our development: annotations, function inlining, better support for several C constructions, etc. Overall, most changes are beneficial, even if some of them imply discarding partial developments in order to embrace solutions made possible by the recent features

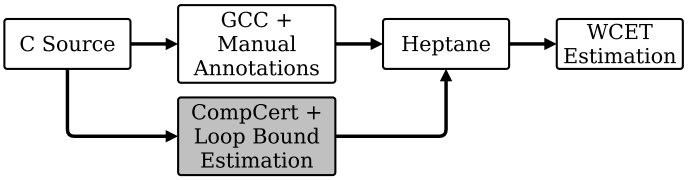


Figure 7.4: Integration of the loop bound analysis into the Heptane WCET estimation tool. By default, Heptane uses GCC to compile the C source and requires manual loop bound annotations. Using our loop bound estimation, we were able to automatically produce annotations in the assembly code, feeding them to Heptane.

(e.g. replacing interprocedural semantics with syntactic function inlining) or updating some proofs due to modifications in the code base. Even more so than in traditional non-verified development, updating the code and the proofs to a new version is a significant undertaking, especially considering dependencies related to Coq (i.e. newer CompCert versions require newer Coq versions, and between them some proofs may fail due to change of behavior related to the application of proof tactics).

All in all, integrating analyses into CompCert provides a way to benefit from several components of the compiler, such as parsing and generation of an intermediate representation, enabling the handling of C source with a limited effort. The overall architecture of the compiler is kept simple, due to the fact that each line needs to be proved. On the other hand, components must reuse the data structures defined in CompCert to avoid re-implementing most of the elements. Finally, CompCert provides a useful framework for experimental evaluation; by accepting C code as input, analyses can be evaluated on several existing benchmarks.

7.2.4 Integration with Heptane

The Heptane [15] static WCET estimation tool performs extensive low-level analysis of binary programs and applies IPET to produce a WCET estimation. However, its flow analysis, which includes CFG reconstruction from the binary, does not include automatic loop bound estimations: loop bounds need to be manually inserted as annotations in the program (either in the source code, if Heptane is configured to compile it, or directly in the assembly code). When producing the ILP constraints, if Heptane detects that a given loop does not have annotations, it stops the analysis, since it knows that the program will not have a finite WCET bound.

We integrated our loop bound estimation in Heptane, by replacing its standard compilation chain, which uses GCC and requires manual annotations, with CompCert and the loop bound estimation, as indicated in Figure 7.4. From the annotated assembly code, Heptane is capable of reconstructing the control flow graph, producing the binary program, and computing a WCET estimation based on a much more sophisticated hardware model than ours (including cache and pipeline analyses).

The main objective of this integration was to validate the usefulness of the loop bound estimation. Since Heptane does not produce loop bounds, no evaluation of the precision of the method was performed, but we managed to confirm that our loop bounds can be used by Heptane to replace manual annotations. Heptane is not formally verified, yet the addition of a verified, automatic loop bound estimation improves confidence in the results given by Heptane, while minimizing the amount of manual effort.

7.3 Experimental Evaluation

We present in Section 7.3.1 the evaluation of the loop bound estimation method (presented in Chapter 4), and in Section 7.3.2 an evaluation of the WCET estimation method based on IPET (described in Section 7.1). Both evaluations have been performed on the Mälardalen WCET benchmarks [34], a reference benchmark for WCET estimation tools. This benchmark provides a set of programs with representative loops, used mainly by WCET tools but also by static analyzers [36]. Its focus on flow analysis makes it a reference for WCET-related loop bound estimations, and well

suited for our tool.

7.3.1 Results of the Loop Bound Estimation

We ran our loop bound estimation method on the Mälardalen WCET benchmarks and obtained the results displayed in Figure 7.5. They have been compared to the results of the loop bound estimation of SWEET [27]. The programs considered are those analyzed in [27] for which SWEET could estimate at least one bound, excluding one that CompCert cannot compile due to an unstructured `switch` statement (program `duff`, which contains a Duff's device).

The second column in the figure (#L) displays the number of loops in each program. The third column shows the accuracy of our estimation of local bounds: it gives the number #LE of estimations of local loop bounds (and their percentage) that are exact bounds. Unfortunately, this column is not given in [27], but we have estimated it from the results of our tool and our manual analysis to infer which loops are estimated by SWEET.

Our results are close to those obtained by SWEET. On average, 74% of the loops are exactly estimated by our method, while 82% of the loops are exactly bounded by SWEET. The histogram in Figure 7.5 shows, for each program, the total number of loops (baseline in light gray), followed by the number of exact local bounds for our tool (in dark gray) and for SWEET (in black). The last two columns of Figure 7.5 give the number `#GB` of loops having been globally bounded (and their percentages). A loop is considered bounded only when it has a meaningful estimation (i.e. different from `MAX_INT` for instance). Note that a loop may be *locally* but not *globally* bounded, if it is nested within a loop which has not been bounded itself. On average, our tool estimates almost as many global bounds as SWEET. Indeed, 82% of global bounds are estimated by our tool, and 86% of global loops are estimated by SWEET.

Differences in precision come from our value analysis, that is slightly less precise than SWEET's. As our value analysis does not handle floating-point values or global variables, nor perform a pointer analysis, 17 loops are bounded by SWEET and not by our method.

Concerning the analysis time, hardware differences make it difficult to compare them with SWEET's. Nevertheless, we could verify that the use of *a posteriori* validation does not incur a significant overhead in our analysis. Benchmarking the programs in Figure 7.5 using a current personal computer takes less than a minute.

7.3.2 Evaluating the WCET Estimation

We also evaluated our WCET estimation technique on the Mälardalen WCET benchmarks, on the same set of 20 programs used for the evaluation of the loop bound analysis. One of the prerequisites for estimating the WCET is that every loop in the program must be bounded, therefore 5 of these programs were not evaluated: `adpcm`, `fft1`, `fir`, `insertsort` and `ludcmp`. Some of them contain loops with floating-point variables (such as `fft1`), others depend on memory contents (such as `insertsort`). In all cases, it is due to imprecisions in the value analysis. For the remaining 15 programs, we compared our estimation to the result of a *reference interpreter* that we implemented from CompCert's formal semantics of the PowerPC assembly. This interpreter is given a program which is known to correspond to the worst-case path and its execution trace produces the exact result of the program execution, which enables us to compute the exact WCET. Our comparisons present the results of the estimation both with and without some semantics-preserving loop transformations (loop inversion and loop unrolling, detailed in the following), whose purpose is to improve the precision of the result. Our evaluation confirmed that these transformations did improve the result of the WCET estimation.

Loop Transformations

During evaluation, we observed that some programs could benefit from loop transformations to produce a tighter WCET estimation. In particular, we implemented and evaluated the improvements in precision by applying *loop inversion* and *loop unrolling*. We describe precisely how we performed these transformations, why they improved the precision of the result, and how their correctness can be proved.

Program	#L	Our tool		SWEET		Our tool		SWEET	
		#LE	%LE	#LE	%LE	#GB	%GB	#GB	%GB
1 adpcm	27	13	48%	22	81%	16	59%	18	67%
2 cnt	4	4	100%	4	100%	4	100%	4	100%
3 cover	3	3	100%	3	100%	3	100%	3	100%
4 crc	6	4	67%	6	100%	6	100%	6	100%
5 edn	12	9	75%	11	92%	12	100%	12	100%
6 expint	2	2	100%	2	100%	2	100%	2	100%
7 fdct	2	2	100%	2	100%	2	100%	2	100%
8 fft1	29	3	10%	6	21%	7	24%	7	24%
9 fibcall	1	1	100%	1	100%	1	100%	1	100%
10 fir	2	1	50%	1	50%	1	50%	2	100%
11 insertsort	2	1	50%	1	50%	1	50%	1	50%
12 jfdctint	3	3	100%	3	100%	3	100%	3	100%
13 lcdnum	1	1	100%	1	100%	1	100%	1	100%
14 ludcmp	11	6	55%	6	55%	6	55%	6	55%
15 matmult	7	7	100%	7	100%	7	100%	7	100%
16 ndes	12	12	100%	12	100%	12	100%	12	100%
17 ns	4	4	100%	4	100%	4	100%	4	100%
18 nsichneu	1	1	100%	1	100%	1	100%	1	100%
19 qurt	3	2	67%	3	100%	3	100%	3	100%
20 ud	11	11	100%	11	100%	11	100%	11	100%
Geometric mean			74%		82%		82%		86%

Figure 7.5: Exact local header bounds and meaningful global bounds obtained on the Mälardalen benchmark. The numbers of loop bounds are given relative to the total number of loops.

Loop inversion Loop inversion [53] consists in replacing `while` and `for` loops with an equivalent `do-while` loop. This means moving the loop condition from the beginning to the end of the loop body. This transformation is simple to perform and has two benefits in terms of performance:

- only one branch instruction is executed when the loop is exited; in a `for` or `while` loop, at the end of the last iteration, the code is branched to the beginning of the loop (before the loop condition) and then branched again to exit the loop. In a `do-while` loop, the last iteration only has one branching, which leads to outside of the loop.
- if the initial condition is statically determined (which is often the case, e.g. most `for` loops initialize the iteration variable with a constant value), then the test before entering the loop can be suppressed.

In terms of number of cycles saved by the optimization, loop inversion does not provide a significant improvement in performance when compared to other optimizations, but its simplicity is an advantage: loop inversion is cheap to apply, which justifies doing it even for small gains. Figure 7.6 presents four equivalent loops, the first and the second ones before loop inversion, and the third one after applying it. In this example, as it often happens in real programs, the outer `if` can be simplified by constant propagation (which gives $0 < 10$) and subsequently removed from the program, resulting in code equivalent to the one in the fourth column. While the source code may seem more complex at first, the assembly code is at least as efficient, even when the outer `if` cannot be simplified.

Our interest in loop inversion is not motivated by performance gains; in our case, loop inversion provides extra precision for the value analysis. Thanks to the extra block created by the outer `if` condition (third column in Figure 7.6), the domain of the loop iteration variable inside the inverted

```
                                  i=0;
                                  if (i<10) {                 i=0;
for (i=0; i<10; i++) {    i=0;      do {                      do {
  body;                   while (i<10) {   body;              body;
}                           body;          i++;               i++;
                            i++;           } while (i<10);    } while (i<10);
                          }                }
```

Figure 7.6: Example of loop inversion in C code. The first and second columns illustrate equivalent `while` and `for` loops. The third column shows the equivalent inverted loop, while the last column presents the same inverted loop after constant propagation and elimination of the unnecessary test.

loop is smaller than in the original loop. For instance, the domain of variable i in Figure 7.6, at the program point of the loop condition, is $[0, 10]$ in the non-inverted loops, and $[0, 9]$ in the inverted one. This difference prevents a small loss of precision in the bounds estimation of these loops. A small imprecision is magnified when there are nested loops and can have a significant impact in the final estimation.

In CompCert, the best language where this transformation can be performed is during the compilation from CompCert C to Clight. Lower-level languages such as Csharpminor have a single kind of loop, which complicates the application of loop inversion. To prove its correctness, a necessary step is to show that the generation of the corresponding `do-while` code (with the extra `if` for the first iteration) results in a semantically-equivalent loop.

Loop unrolling Loop unrolling consists in manually unfolding some loop iterations, for instance by copying the body of the loop along with its exit conditions. As many copies as desired can be made, without changing the semantics of the program: after the execution of each copy of the body, the loop condition is executed, and the loop is exited if the condition is not met. Figure 7.7 presents an example of loop unrolling in C code. The first column displays the original program, the second column displays the same loop with its exit condition rewritten (to more closely match the behavior of the unrolling), and the third column depicts the actual unrolling. In this example, we arbitrarily unrolled the loop three times, resulting in four copies of the loop body. The last copy of the loop is unnecessary here. In fact, it corresponds to dead code, since the previous condition is always true. Even so, having this extra copy does not affect the correctness of unrolling: the program behaves just like before unrolling.

Loop unrolling is usually performed to improve the performance of a program. This is achieved by eliminating some branching conditions between copies of the loop, which improves, among others, pipeline performance. For instance, in Figure 7.7, a value analysis can infer that the first two conditions statically evaluate to $(!(1 < 3))$ and $(!(2 < 3))$, which are both false, and therefore these tests can be removed. In our case, loop unrolling improves the precision of the WCET estimation in some programs, namely those having a structure such as this one:

```
i = 0;
do {
  if (i < 2) { expensive computation; }
  i++;
} while (i < 5);
```

In this program, the body of the `if` contains an expensive computation which is executed only a few times. However, the WCET estimation will pessimistically consider that the `if` branch may be taken at every loop iteration. This will lead to a loss of precision. One way to prevent this is to add flow facts relating the program points inside the `if` and those outside, but the automatic generation of such flow facts is a complex task. Another way is to modify the value analysis to perform some form of "virtual unrolling", allowing a single program point to correspond to several states in the analysis. This is a very expensive modification which requires changing the entire structure of the value analyzer and imposes a heavy proof burden. Yet another way to improve

```
i = 0;
do {
  body;
  i++;
} while (i < 3);
```

```
i = 0;
do {
  body;
  i++;
  if (!(i < 3)) break;
} while (1);
```

```
i = 0;
do {
  body;
  i++;
  if (!(i < 3)) break;
  body;
  i++;
  if (!(i < 3)) break;
  body;
  i++;
  if (!(i < 3)) break;
  body;
  i++;
  if (!(i < 3)) break;
} while (1);
```

Figure 7.7: Example of loop unrolling in C code. Before unrolling the loop body in the first column, it makes more sense to consider the loop as written in the second column. The third column depicts the loop after unrolling it 3 times. The last copy of the body is actually dead code, but this does not affect the behavior of the program.

precision is to use loop unrolling, which syntactically expands the loop body and allows a "standard" value analysis to infer precise information at each (expanded) program point. For instance, unrolling the previous loop at least 2 times is sufficient for the value analysis to detect that no loop iteration after the third one will execute the code in the conditional branch. The resulting program and ILP constraints will be precise with respect to the conditional branch.

An advantage of loop unrolling, in terms of proof, is that is is semantically correct to choose any value for the unrolling factor, therefore it is not necessary to prove the correctness of any particular heuristics chosen for the unrolling factor. On the other hand, it is necessary to prove the semantic preservation for *any* value.

We chose to perform loop unrolling during the compilation of Csharpminor code into Cminor code. Both languages have a single loop structure, which minimizes the proof effort, and Cminor is the last structured language of the compilation chain; performing loop unrolling in a lower language would require a loop reconstruction algorithm and the proof of some properties related to it.

Concerning the development, the most complex part of loop unrolling is the definition of the heuristic function, which has to balance precision and efficiency, avoiding excessive unrolling. Experimental evaluation has been essential in this part, to measure whether unrolling was worthwhile, and if its cost (in terms of increased program size) was reasonable. The chosen heuristics was based on several factors: as a first approximation, each loop is completely unrolled to obtain maximum precision. In other words, each loop is unrolled as many times as the loop bound estimation believes it will execute. If it eventually executes less iterations, the only consequence is that some dead code will be generated. There are, though, some exceptions which help minimize the size of the final program: first, loops are not unrolled unless they actually contain conditional branches. Otherwise, unrolling does not contribute to precision. Second, some arbitrary limits have been set on the maximum number of iterations, to avoid unrolling loops with extremely high iteration counts. In the end, we manage to limit code explosion due to unrolling while improving precision in some programs, as will be presented in Section 7.3.2.

Reference Interpreter

The reference interpreter of the assembly language takes a CompCert assembly[2] program as input and executes it step by step, using the executable formal semantics of the language to evolve its

[2]Our current implementation uses the PowerPC assembly semantics, but it can easily be adapted for the x86 or ARM variants of CompCert.

state in accordance with the expected behavior of an actual machine. Note that the input program is the memory representation compiled by CompCert, and not an actual `.s` file[3].

CompCert's formal semantics does not include timing information, but the reference interpreter can reuse the same hardware timing model defined for the WCET estimation. This ensures the coherence between the exact WCET given by the interpreter and the one given by the WCET estimation. Using the interpreter is an advantage over using an external simulator that does not share the same formal semantics: there are no semantic mismatches to be taken into account.

To correctly use the reference interpreter, it is necessary to know which program inputs trigger the worst-case path. These inputs must be given to the interpreter, otherwise it may compute a value lower than the actual WCET. This leads to the impression that the WCET estimation is less precise than it actually is. Note that the program given to the interpreter is not necessarily identical to the program under analysis: the former has some predefined inputs to trigger the worst case, while the latter is more general.

After determining the worst-case paths in our set of benchmarks, we instrumented them for the interpreter (i.e. setting the inputs to activate the worst case) and ran each program to obtain the exact WCET. We then compared it to the estimated WCET given by our analyzer.

Results

Figure 7.8 presents the results of our evaluation. For each program, we indicate the size of its source code (*LoC*) and we present the relative overestimation $\left(\frac{\text{our WCET - exact WCET}}{\text{exact WCET}}\right)$, using three different configurations: without using any loop transformations, using loop inversion only, and using loop inversion together with unrolling, where the unrolling factor is calculated for each loop using the heuristics described previously. We also present the execution times of our tool.

		No Loop Transformations		*Loop Inversion*		*Inversion+Unrolling*		
Program	***LoC***	***Overestimation***	***Time (s)***	***Overestimation***	***Time (s)***	***Overestimation***	***Time (s)***	***Class***
cnt	267	18.3%	0.1	2.8%	0.2	3.3%	7.0	OK
cover	640	10.9%	1.0	11.5%	1.0	0.0%	21.8	OK
crc	128	100.2%	0.2	99.5%	0.2	99.2%	1.7	Imprecise
edn	285	141.5%	12.5	110.4%	13.1	110.4%	23.4	Imprecise
expint	157	2601.6%	0.0	2419.7%	0.0	0.0%	8.2	OK
fdct	239	0.0%	0.4	0.0%	0.5	0.0%	0.6	OK
fibcall	72	0.9%	0.0	1.1%	0.0	1.1%	0.0	OK
jfdctint	375	0.0%	0.3	0.0%	0.3	0.0%	0.5	OK
lcdnum	64	50.9%	0.0	55.2%	0.0	11.9%	0.1	OK
matmult	163	11.5%	0.3	0.0%	0.3	0.0%	0.5	OK
ndes	231	12.2%	4.0	3.6%	4.2	3.6%	225.4	OK
ns	535	88.3%	0.1	0.2%	0.1	0.2%	0.2	OK
nsichneu	4,253	106.1%	60.5	106.1%	60.2	106.3%	89.7	Imprecise
qurt	166	168.2%	0.7	165.7%	0.7	215.2%	3.0	Imprecise
ud	161	225.1%	0.6	217.3%	0.6	265.2%	11.3	Imprecise

Figure 7.8: Experimental results of our WCET estimation, given as a relative overestimation $\left(\frac{\text{our WCET - exact WCET}}{\text{exact WCET}}\right)$. Results are presented without and then with loop transformations.

We classify the programs in two groups: OK (WCET estimation with no or little overestimation) and Imprecise (significant overestimation). The main source of imprecision is the value analysis: several programs depend on floating-point (e.g. `qurt`) or memory variables (e.g. `crc`) not currently tracked.

In most programs, loop transformations improve precision, sometimes drastically (e.g., `ns` goes from 88% down to 0% overestimation thanks to loop inversion, and `expint` goes from 2,420% to 0% due to unrolling). `ns` is an illustrative example of loss of precision in nested loops, as mentioned in Chapter 4: this program contains a quadruply-nested loop in which off-by-one imprecisions in the local loop bounds result in a global bound two times larger than the exact bound.

In a few programs we see a *relative* increase due to the decrease in the *absolute* WCET of the transformed loops. For instance, in `qurt`, the interpreter computes a WCET of 576 instructions

[3]A parser could be used to reconstruct the program given a pretty-printed file, but the formal guarantees given by CompCert would be lost.

for the original program, but only 376 instructions for the loop-optimized version. The WCET estimation is also improved, from 1,553 cycles to 1,185 cycles, but its improvement is proportionally smaller. Further improvements to the WCET estimation may come either from a technique to obtain bounds for triangular loop nests (for `ud`), or from a more sophisticated value analysis (for the other programs).

7.4 Conclusion

Our formalized development resulted not only in a tool to obtain WCET estimations, but in a general-purpose static analysis framework with several parameters and modes of operation. Besides having been used during the development, as debugging tools and for the experimental evaluation, these analyses are helpful for extensions and developments which are not necessarily WCET-based. Overall, we consider that a formal development is not limited to the parts which have been entirely formalized, but it also benefits from the surrounding development which enriches it, without being detrimental to its correctness.

Concerning the WCET pen-and-paper formalization, it presents the main ideas of the proof and is inspired by work on result certification. The experimental setup enabled us to confirm that the results of our WCET estimation tool are useful; it also showed that, thanks to the integration with the CompCert compiler, there are several opportunities to apply transformations (such as loop inversion) to obtain more precise results while reducing the proof effort. Finally, the reference interpreter provided a means to evaluate the WCET estimation with precision.

About the experimental evaluation, from the results obtained we noticed that most improvements would come from a more sophisticated value analysis, integrating some sort of memory inspection. Even a simplified global variables analysis could bring extra precision. Floating-point and strided intervals are also useful for some of the loops, such as those with non-unit increments, and their implementation could improve precision in a few cases. We also observed that several programs benefit from simple loop optimizations, and thanks to the integration with the CompCert compiler we could quickly implement and test them.

Chapter 8

Conclusion

A safe, trustworthy WCET estimation is necessary for the development of safety-critical, real-time software. Formal verification methods attain very high levels of confidence in software development, and their application in industrial-scale tools is maturing. A complete formal verification of a WCET estimation tool, from the source code down to the executable level—including the precise modeling of modern hardware architectures—still eludes researchers and developers alike. In our approach, we started with the upper layers, focusing on issues related to the source code, the programming language, and its compilation into assembly code. Several issues have been resolved at this level, and from our formal development we have been able to derive some verified static analysis tools of broader applicability than WCET estimation alone.

One of the goals of a formal development, besides providing better understanding of the formalized tools and techniques, is to minimize and strengthen the trusted computing base (TCB). One of the main vulnerabilities of static WCET estimation tools is the integration of manual annotations related to control flow information, namely loop bounds. Such annotations are fragile and constitute a major weakness of the TCB; replacing them with formally verified, automatically generated annotations is a major improvement in terms of trustworthiness.

Loop bound annotations constitute only a fraction of the WCET estimation process. It largely depends on the compilation process, which produces the low-level machine-dependent code from the source written by the programmer. While the formally verified CompCert compiler produces bug-free code, supported by a semantic preservation theorem, it does not provide WCET-related guarantees. The work in this thesis provides a first step into extending this correctness theorem to include WCET-related information.

Finally, from the point of view of software engineering, formal development techniques are being increasingly applied to high-assurance software, but they are still considered sometimes as being extremely complex and costly, due to the proof efforts involved. We provide several examples of lightweight proof techniques, based on verified *a posteriori validation* and on the combination of different analyses, which provide strong guarantees at a reduced proof cost. We also constantly perform experimental evaluation of our developments, to complement them with evidence of actual applicability. We believe that such examples help encourage the application of formal methods.

We summarize contributions of this thesis, separating them into formal and experimental aspects. Afterwards, we present some perspectives for future work.

8.1 Formal Contributions

The formal verification contributions of this thesis are concentrated in three main subjects: loop bound estimation, program slicing and IPET formalization. The first two have been mechanically verified in Coq, while the latter has only been structured on paper. We present each contribution in turn, focusing on the lessons learned from each formalization effort. In terms of size of the verified development, both program slicing and the loop bound estimation each represent about one man-year of proof effort. The mechanized formalization of the value analysis, mainly developed by three other people in the team, consists of about 1.5 man-years of proof effort.

Loop Bound Estimation The semantic justification of a loop bound estimation involves several terms which are intuitively understood as "simple", though in practice they involve subtle details requiring some formalization effort. The very notions of loops, loop bounds, and loop bound estimation (by counting different variable states) each require a precise definition, based on a counting semantics, and non-trivial algorithms to compute them.

The loop bound estimation is the best example of inter-cooperation between the different intermediate languages in the CompCert compiler: before arriving at the verified loop bounds at the assembly level, it is necessary to consider (1) the high-level structured C representations and the loop inversion performed at this stage (which prevents later issues with off-by-one bounds imprecision); (2) the Cfg language, where the loop bound estimation is actually performed; and (3) the assembly language, which receives valid bounds thanks to the semantic preservation guaranteed by CompCert's annotations. Without the different compilation stages, it would be much harder to compute and prove correct a loop bound estimation directly at the assembly level.

The formalization of the loop bound estimation also allowed us to realize the importance of a pragmatic approach concerning at the same time proof and code generation. Indeed, although we present the contributions separately, in reality their mutual influence affects both parts and is visible in the proofs and algorithms: some proofs become obsolete due to validation, and extraneous computations are added to facilitate some proofs. A noteworthy example is the use of loop inversion to improve the precision of the loop bound estimation, while maintaining existing proofs.

Program Slicing Program slicing has been extensively studied and has existing formal verification efforts [67], but they are based on relational (non-executable) specifications and are directed towards different applications than our loop bound estimation. For this reason, their results are not automatically convertible into executable algorithms, or require substantial modifications to the formalization. One of the reasons is the complexity of the techniques involved in program slicing, such as the computation of postdominance and control dependencies, which require sophisticated algorithms and data structures (e.g. computation of Tarjan's connected components) to be computed efficiently. Our approach consisted in using *a posteriori* validation to obtain a solution providing an equivalent level of confidence in terms of correctness. By verifying the *result* of program slicing, our approach frees the programmer of several restrictions concerning the computation of the program slice, as long as enough information is given to the validator (in our case, such information is given by the sets of *relevant variables* and *next observable vertices*).

IPET Formalization Our pen-and-paper formalization of the IPET, plus an experimental evaluation of its results, gave us useful insight into the most relevant aspects of the WCET estimation and enabled us to confirm the feasibility of our approach. The experience acquired with the formalization of the loop bound estimation (which shares some common aspects with this one, such as the use of an instrumented counting semantics), plus the fact that Farkas' lemma has already been formalized and validated in Coq before ([8]), plus the fact that no other components depend on the IPET led us to consider this proof as of lesser priority, since its results would not impact other components. We validated the experimental results through the use of a reference interpreter, to ensure the precision is acceptable.

The most interesting aspect of the formalization of the IPET is that, while developing the proof, we became aware that some precision issues in the loop bound estimation could be solved by applying transformations which would ultimately simplify the formal proof of the WCET estimation. In other words, formalizing one problem helped to solve a somewhat unrelated issue in another part of the development. This is not an exceptional situation, and one of the reasons why formalization is useful: it forces us to reason globally and to keep the development as straightforward as possible.

8.2 Experimental Contributions

Correctness is not the only important characteristic in our verified development: completeness, precision and efficiency are important when considering the usefulness of a piece of software, and the most efficient way to evaluate them is through experimentation. We performed a number of different evaluations, each of them important for a specific part of our development. We include the

tools themselves as an experimental contribution, since they are both necessary for reproducing the results and useful for performing future evaluations. The experimental contributions of this thesis are: the evaluation of the value analysis; the evaluation of the loop bound and WCET estimations; and the implementation of the formalized analyses (and accessory transformations) to be used independently of the main computations. These contributions, available online, can be used to re-run the experiments (except for the external tools used in the value analysis comparison, i.e. Frama-C and Wrapped). Each of these contributions, detailed in the following, brought new insights and often unexpected results, which imposed reconsideration of developments already underway.

Evaluation of the value analysis The lack of standard benchmarks in the static analysis community led us to devise an experimental protocol for the comparison of different analyzers. These analyzers, while similar in their results, contain a significant number of differences which must be taken into account during the comparison. We summarize the main observations from the experimental setup related to the value analysis:

- comparing value analyses (and several other static analyses) requires considerable effort and is not scalable (each analysis requires specific considerations);
- the development of a value analysis should be guided by realistic examples;
- a single kind of benchmarks is not sufficient.

One of the difficulties in the comparison is the granularity of information: a value analysis produces too much information, and much of it is not directly comparable. It would require that both analyses operate at the same level, using the same variables and program points. Since no analysis actually operates at the C source—each uses its own internal representation, which is at best a rewritten version of the C code—their comparison requires non-trivial considerations and metrics.

Benchmarks for static analysis with a specific application in mind, such as "detection of out-of-bounds array accesses for security concerns", are more common (e.g. the Verisec suite [41]), but they only evaluate the value analysis indirectly and in a partial manner. We adapted CompCert's benchmarks to obtain a more direct evaluation, but paying attention not to collect too many data points. This required experimenting with several parameters (e.g. program points to consider, variables to take into account) to obtain a solution giving sufficient (but not excessive) information and also scalable (not requiring too much manual work for each added benchmark). However, our approach only serves to perform relative comparisons between methods, without the possibility of measuring the *absolute* precision of the analysis. Such an evaluation would require a considerable amount of manual work and would not scale well for larger programs.

The results of the evaluation indicate that experimental feedback is essential to develop an analysis for a language as expressive as C (including dozens of operators and language constructs): most variables can be bounded by dealing with a specific subset of the language operators. Experimentation helps finding which operators are needed (e.g. integer addition is performed by different operators, depending on the architecture), and which operators provide too little benefit relative to the cost of their integration in the analysis (e.g. some bitmask and logical operators are seldom used).

Evaluation of the loop bound and WCET estimations Our loop bound evaluation followed metrics similar to those of existing estimation tools: the most relevant values are (1) number of bounded loops, and (2) bounds precision. The main factor influencing our loop bound estimations is the value analysis, which offers several possibilities for improvement (such as the addition of more domains). Another important element is our of definition *program variables*, which is simplistic with respect to the memory: seeing it as a "single huge block" introduces imprecision, and therefore some loops are left unbounded. *Alias analyses* such as Robert and Leroy's [64] can restore some precision, at the cost of introducing profound changes to most analyses, since they depend on this definition themselves.

Obtaining an experimental WCET estimation allowed us to validate that no components have been forgotten in the WCET estimation chain. Although a hardware model would help obtaining a

more realistic WCET, the current IPET constraints based on our loop bound estimation are sufficient to bound a number of programs with satisfactory results. We provided a reference interpreter for the PowerPC assembly language, which has potentially other uses besides our WCET estimation. This evaluation also prompted us to perform a more thorough integration of the entire CompCert chain: instead of operating mostly at a single intermediate language (Cfg), we introduced loop transformations (performed in other languages) to improve on the WCET estimation.

One aspect which became clear when performing the WCET estimation was the lack of open-source, conceptually simple hardware models which could be "plugged" as cost coefficients in the ILP. Hardware models are much more specific than high-level analyses, which means the work is less rewarding, since it has a narrow applicability. This means they tend to be performed more often in commercial WCET tools than in research ones; also, due to licensing opportunities, some research institutes do not divulge their low-level work. Finally, for those which do share the source code, it is often the case that the hardware model is tightly coupled to other parts of the analysis—due to necessity, when modeling complex timing dependencies, or for practical reasons—which prevents it from being extracted and reused by other tools. This illustrates the importance of developing a formally verified model for WCET estimation, and also serves as a forewarning of the difficulties which lie in that direction.

Implementation of the formalized tools In our formal development, as it is customary in verified software development, we produced tools besides the ones which have been formally verified. They compute results which are validated *a posteriori*, or they compute intermediate information used between components of the verified analyses and which does not necessarily have a separate correction theorem. In either case, they are a useful by-product of the development. Our several validated components have various uses. For instance, when debugging the origin of some loss of precision in the WCET estimation, we can slice the program and perform a value analysis. We can also experiment with new algorithms and modifications to the analysis before proving them.

A noticeable aspect of this contribution is the fact that the use of validation enables loose coupling between different parts of the code, allowing reuse of external components of several natures (e.g. OCaml code for program slicing, C code for the LP solver). Although in theory some programs might fail validation, in practice it only happens for corner cases which the analysis cannot handle anyway (e.g. a fabricated program with degenerate control flow). Overall, the implementation of the formalized tools has been a benefit not only in terms of "output", but also as input for the development process itself.

8.3 Perspectives

There are several directions worth exploring to advance the work in this thesis. From the high-level analyses down to the machine timing aspects, each direction offers opportunities to improve on the results and to increase the precision of the WCET estimation. We present here an outline of the aspects which we believe should be treated in priority, to improve confidence in the analysis and deal with more realistic configurations.

Conclusion of the formally verified WCET estimation tool The immediate concern is the finalization of the verified WCET estimation. Proving in Coq the verified validator of the LP solver would provide a higher level of guarantee for the entire WCET estimation tool. This would not impact the performance nor the precision of the estimation, and would add an extra property to CompCert's semantic preservation theorem.

Improving existing analyses To handle increasingly complex programs, some of the current analyses can benefit from more sophisticated algorithms. Experience from the Verasco analyzer (and previously from the Astrée analyzer) shows that, in large (realistic) programs, the use of specific abstract domains is necessary to handle with specific parts of the code. A generic solution such as a single abstract domain rarely provides a good cost/precision trade-off. Similarly, when dealing with issues such as program slicing and bounds estimation, for instance, the isolation and transformation of program fragments which might prove problematic (specific kinds of loops, jump

tables, etc.), obtains better results at a reasonable development and proof cost. The integration of new analyses, such as an alias analysis to more finely deal with memory contents, also provides alternatives for improvement.

More precisely, some exploratory experimentation with a formally verified alias analysis [64] demonstrate its improvements in terms of loop bound estimations. The benefit/effort ratio is however relatively limited, due to the amount of necessary modifications. Other results, this time drawn from the experimental evaluation, indicate improvements via the inclusion of *strided intervals* (to handle situations such as loops with non-unit increment, such as `for (i=0; i<N; i+=2)`), small sets in complement to intervals (for short, irregular loops) and relational domains for triangular loop nestings. Inclusion of floating-point intervals is also worthwhile.

Finally, concerning the applicability of the analyses to a broader class of programs (such as programs with irreducible loops), the results observed in this thesis favor the application of program transformations and normalization instead of a reformulation of the proofs to consider the more general class of programs. As it has been observed in CompCert, where the introduction of several intermediate languages leads to simpler proofs, in our experience the introduction of several independent transformations (e.g. node splitting for irreducible loops, compilation of switches as decision trees, loop inversion, etc.) is a much more efficient approach to deal with these situations. As an added bonus, these transformations can have other uses, such as in future optimizations and analyses.

Improving flow fact generation Loop bounds are the fundamental flow fact for WCET estimation, but far from being the only one. The generation of additional flow facts improves the precision of the estimation and is essential for industrial WCET tools, where overestimations are ideally in the range of single percentage digits. Flow facts are often manually inserted, making them part of the trusted computing base. Automatically generating them, or at least being able to verify their correctness, is an important step in the development of formally verified industrial WCET tools. We experimented with some extra constraint generation derived from the value analysis, e.g. by limiting relative execution frequencies of `if` branches inside a loop. We observed some improvement in specifically crafted examples but not in the benchmarks, because there the particular situation which enabled the generation of these flow facts did not happen. Using some form of virtual unrolling, it might be possible to broaden the applicability of such flow facts. A mechanism to validate flow facts *a posteriori* (e.g. using some techniques derived from symbolic execution applied to WCET analysis [9]) seems a worthwhile approach. Flow facts could be produced by untrusted code and given as a heuristic for the (verified) validator. This way, several kinds of flow facts could be produced at little cost. The validation would probably be more costly than the one used to validate the LP result, but the genericity of the flow fact producer would be a great advantage in terms of proof effort.

Towards more realistic hardware models The current timing model is very simplistic and does not correspond to modern architectures. It is currently defined as a mapping between instructions and the (constant) number of cycles their execution takes. This is realistic for simple microcontrollers, such as the 8051, but for more advanced processors at least a cache or a pipeline analysis are necessary. These static analyses are common in WCET tools, and their integration would be useful to broaden the range of high-confidence components in a verified WCET estimation tool.

To integrate a cache (or pipeline) analysis, the behavior of the component must be formally specified (increasing the trusted computing base). Then, a data-flow framework (such as the one present at the Cfg level in CompCert) enables the definition of an abstract interpretation-based analysis which computes an over-approximation of the cache (or pipeline) contents, in order to predict cache hits (or pipeline speedup). Also, despite efforts to isolate hardware-related aspects in a single part of the WCET estimation process, it is well-known by experts in the field that adding more aspects of the hardware behavior tends to impact the entire architecture of the tool. Formalizing these changes plus the additional IPET constraints related to them requires a gradual incorporation of features, but it is a promising avenue for exploration.

Real-world benchmarking Even after implementing several of the previous modifications, our analysis would not be complete without trying it on real programs. The WCET benchmarks provide interesting use cases, but they are not representative of all the issues present in actual development software. We performed some tests of the value analysis on a real code base for the PHEBUS spectrometer [14], part of the ESA project BEPI COLOMBO, which has sent a probe to study the atmosphere of Mercury. This allowed us to better understand some of the issues present in realistic code bases, such as some inevitable complex pointer manipulations, implementation-defined behaviors, and scalability issues. In the medium term, a (partially) verified WCET estimation of such a code base is feasible, with the addition of some form of manual annotations for critical code sections whose semantics are not necessarily defined in C (e.g. implementation-defined behaviors). These regions are limited in scope and the specific necessary behaviors (e.g. casting a pointer into an integer, and then vice-versa) may be axiomatized. The hardware used in PHEBUS (an FPGA-emulated 8-bit microcontroller), which is also used in other safety-critical systems (such as in the automotive industry), is sufficiently simple so that a WCET estimation would not have much difficulty handling the hardware model. The main difficulty would come from obtaining precise flow facts.

Afterwards, the benchmarking objectives would include code bases for more advanced processors, such as the MPC755 PowerPC used by Airbus as a flight control computer. The C code base produced by code generation from SCADE models provides an excellent use case and real-life benchmark for our WCET estimation. The integration of the CompCert compiler in this code base [31] has already been evaluated, and the WCET estimated by a non-verified tool (aiT) indicated improvements with respect to the current tool. Formally verifying a WCET estimation technique would improve confidence in the results. However, cache and pipeline analyses are necessary here, as well as a much more scalable analysis due to the size of the code base. On the other hand, it is probable that loop bound estimation will be less demanding, due to coding conventions which avoid using loops. Succeeding in performing this formal verification may be considered as the major medium-term research goal for the work in this thesis. The long-term goals are similar to those of the CompCert compiler: to bridge the gap between hardware verification efforts and verified software, generating a continuum of verified components conform to their specification.

Index of Notations

Appendices

Appendix A

Syntax of the CompCert Cfg Language

The abstract syntax of the CompCert Cfg language is presented in Figure A.1 below. It is a slightly simplified version of the actual Cfg syntax, with some elements irrelevant to this thesis omitted.

Numeric constants are either integers or floating-point values; other constants include symbol (variables, functions, etc.) and stack addresses, with optional offsets. Unary and binary expressions include all of C operators (arithmetic, logical, bitwise, ternary operator, etc.). Memory loads are also expressions. Arithmetic operators are not overloaded concerning signed and unsigned types: both versions exist.

Each statement in Cfg has labels (l, l_{true} and l_{false}) indicating its successors. Statements include assignments, memory stores, conditional branches, function calls and function returns. Function definitions include internal functions, defined within the program being compiled, and external functions, including compiler *built-ins* and functions whose code is not available (e.g. code from other compilation units). A function definition includes its signature (list of parameters), its code (as a mapping from program points to instructions), and its entry point. A program is a list of function definitions and global variables.

Program fragments in Cfg are written in a C-like syntax, with special notation to avoid lengthy variable names. For instance, `addrsymbol`(id,n) is written simply as `(`*`id + n`*`)`, and `addrstack`(n) is written as `&`*`n`*. Assignments are written using the traditional `=` symbol, and memory accesses are written inside square brackets, prefixed by the type of addressing (indicated as κ in Figure A.1). For instance, `assign`(x,`load`(`int32signed`,`addrstack`(8)),l) is written as `x = int32signed[&8]`. The successor of the current program point, l, is only written explicitly when it is not the program point in the following line. In this case, we note it by `goto` *`l`*.

Constants:	$c ::= n$	integer constant
	$\mid f$	floating-point constant
	$\mid \texttt{addrsymbol}(id, n)$	address of a symbol plus an offset
	$\mid \texttt{addrstack}(n)$	stack pointer plus a given offset
Expressions:	$e ::= id$	variable identifier
	$\mid c$	constant
	$\mid op_1\ e$	unary operation
	$\mid e_1\ op_2\ e_2$	binary operation
	$\mid e_1 ?\ e_2 : e_3$	conditional expression
	$\mid \texttt{load}(\kappa, e)$	memory load
Unary operators:	$op_1 ::= \texttt{boolval}$	0 if false/null, 1 otherwise
	$\mid \texttt{negint}$	integer opposite
	$\mid \texttt{notbool}$	boolean negation
	$\mid \texttt{notint}$	bitwise complement
Binary operators:	$op_2 ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$	arithmetic integer operators
	$\mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{\|} \mid \texttt{\^{}}$	bitwise operators
	$\mid \texttt{/}_u \mid \texttt{\%}_u \mid \texttt{>>}_u$	unsigned operators
	$\mid \texttt{cmp}(b)$	integer signed comparisons
	$\mid \texttt{cmpu}(b)$	integer unsigned comparisons
Comparisons:	$b ::= \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=}$	relational operators
Statements:	$i ::= \texttt{skip}(l)$	no operation (go to l)
	$\mid \texttt{assign}(id, e, l)$	assignment
	$\mid \texttt{store}(\kappa, e_{\text{dest}}, e_{\text{src}}, l)$	memory store
	$\mid \texttt{if}(e, l_{\text{true}}, l_{\text{false}})$	if statement
	$\mid \texttt{call}(id^?, e_{\text{fn}}, e_{\text{arg}}*, l)$	function call
	$\mid \texttt{return}(e)^?$	function return
Functions:	$fn ::= \texttt{intern}(id*, l \mapsto i, l_{\text{entry}})$	function definition (parameters, code, entry)
	$\mid \texttt{extern}(id*)$	reference to external function (parameters)
Programs:	$p ::= (id \mapsto fn, id_{\text{main}}, id_{\text{gvar}}*)$	program definition (functions, main function, global variables)

Figure A.1: Abstract syntax of the Cfg language. $s^?$ denotes that s is optional. $s*$ denotes an arbitrary $(0..N)$ number of occurrences of s. $a \mapsto b$ denotes a mapping from a to b.

Appendix B

Example Program, from C to Sliced RTL

Figure B.1 presents a small C program, compiled to Cfg and then to RTL, and finally its RTL slice with respect to program point 3 (a loop annotation), to illustrate some aspects of the compilation chain, and also to illustrate how program slicing improves the precision of the loop bound analysis.

In the C program, we have a simple loop with an annotation (e.g. manually inserted by the user) and some computations involving an array. In the end, we return the last element of the array.

In the Cfg code (second column), there is the introduction of program points, the `while` loop is transformed into an `if` plus some `goto`s, and the array variable `a` is transformed into a local stack variable identified by its offset with respect to the beginning of the stack (`&a`).

In the RTL code (third column), local variables are converted into pseudo-registers `x1` to `x6`, and some Cfg expressions are split into several statements. There are some slight simplifications, e.g. `&0 + 4 * 4` is evaluated to `stack(16)`.

Finally, in the last column, we have the result of slicing the RTL program with respect to program point 3, which is the loop annotation. The CFG-preserving slice replaces every instruction associated with the C variables `j` and `a` with `nop`. Note that slicing does not remove program variables *per se*, but several of them (`x2`, `x4` and `x5`) are no longer present in any statement, therefore they are trivially non-interesting. Without slicing, variable `x2` would be considered interesting and taken into account during the loop bound estimation.

```
int main() {
  int i, j, a[5];
  i = 0;
  j = 0;
  do {
    __annot("loop_1");
    j += 7 * i;

    a[i] = j;

    i++;
  } while (i < 5);

  return a[4];

}
```

```
main() {
  local vars: [i, j, &0]
    1: i = 0
    2: j = 0
    3: skip
      4: builtin annot("loop_1")
      5: j = j + 7 * i

      6: int32[&0 + 4 * i] = j

      7: i = i + 1
      8: if (i < 5) goto 9 else goto 10
      9: goto 4
   10: return int32[&0 + 4 * 4]
}
```

```
main() {
  local vars: [x1, x2, x3, x4, x5, x6]
    1: x1 = 0
    2: x2 = 0
    3: nop
      4: x3 = annot "loop_1"()
      5: x4 = x1
      6: x4 = x4 * 7
      7: x2 = x2 + x4 + 0
      8: x5 = stack(0)
      9: int32[x5 + x1 * 4 + 0] = x2
     10: x1 = x1 + 1
     11: if (x1 <s 5) goto 12 else goto 13
     12: goto 4
   13: x6 = int32[stack(16)]
   14: return x6
}
```

```
main() {
  local vars: [x1, x2, x3, x4, x5, x6]
$  1: x1 = 0
     2: nop
     3: nop
@    4: x3 = annot "loop_1"()
     5: nop
     6: nop
     7: nop
     8: nop
     9: nop
$   10: x1 = x1 + 1
$   11: if (x1 <s 5) goto 12 else goto 13
     12: goto 4
   13: nop
   14: return x6
}
```

Figure B.1: Example of a C program compiled into Cfg, then RTL, and finally its RTL slice with respect to program point 3. In the last column, `$` indicates program points in the slice set and `@` is the slicing criterion (also in the slice set).

Appendix C

Status of the Formal Developments

Table C.1 below summarizes the status of each component of the formal development. For each part, we detail the main authors of the specification, the proofs and the experimental evaluation (when relevant). We also present the proof techniques used (if it has been entirely proved within Coq, if *a posteriori* validation has been used, or if the proof is a pen-and-paper one). The final column summarizes the status of the component, including additional details in some cases.

Component	Specif.	Proof	Eval.	Proof tech.	Status
Cfg language & semantics	VER	VER	N/A	Coq proof	C → Cfg compilation done; Cfg → RTL compilation + optimizations not done
ICfg semantics	Me	Me + DP	N/A	Coq proof	Done
RTL loop reconstruction	DP	DP	N/A	Coq proof + validation	Done (adapted from existing development)
Program slicing	Me	Me + DP	Me	Coq proof + validation	Done
RTL value analysis	DP	DP	Me	Coq proof + validation	Done
Verasco value analysis (Cfg)	VER	VER	Me	Coq proof + validation	Done (ongoing work on newer versions)
Loop bound estimation	Me + DP	Me + DP	Me	Coq proof + validation	Done
Loop transformations	Me	Me	Me	Pen-and-paper	Specified (but not yet proved) in Coq
WCET via IPET	Me	Me	Me	Pen-and-paper	Specified (but not yet proved) in Coq

Table C.1: Status of each component of the formal development. DP stands for David Pichardie; VER stands for the rest of the Verasco ANR team.

Bibliography

[1] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[3] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2010.

[4] Roberto M. Amadio, Nicholas Ayache, Yann Régis-Gianas, and Ronan Saillard. Certifying cost annotations in compilers. *CoRR*, abs/1010.1697, 2010.

[5] R. Armadio, Andrea Asperti, Nicholas Ayache, B. Campbell, Dominic P. Mulligan, R. Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, and I. Stark. Certified complexity. *Procedia CS*, 7:175–177, 2011.

[6] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *SEUS*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010.

[7] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language, version 1.5, 2011.

[8] Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. Certified result checking for polyhedral analysis of bytecode programs. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *TGC*, volume 6084 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2010.

[9] Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. The auspicious couple: Symbolic execution and WCET analysis. In Claire Maiza, editor, *WCET*, volume 30 of *OASICS*, pages 53–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[10] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[11] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, pages 324–344, 2013.

[12] Sandrine Blazy, André Maroneze, and David Pichardie. Formal verification of loop bound estimation for WCET analysis. In Springer, editor, *Verified Software: Theories, Tools, Experiments - 5th International Conference (VSTTE 2013)*, Lecture Notes in Computer Science. Springer, 2013.

[13] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of FMPA 1993*, volume 735 of *LNCS*, pages 128–141. Springer-Verlag, 1993.

[14] E. Chassefière, J.-L. Maria, J.-P. Goutail, E. Quémerais, F. Leblanc, S. Okano, I. Yoshikawa, O. Korablev, V. Gnedykh, G. Naletto, P. Nicolosi, M.-G. Pelizzo, J.-J. Correia, S. Gallet, C. Hourtoule, P.-O. Mine, C. Montaron, N. Rouanet, J.-B. Rigal, G. Muramaki, K. Yoshioka, O. Kozlov, V. Kottsov, P. Moisseev, N. Semena, J.-L. Bertaux, M.-Th. Capria, J. Clarke, G. Cremonese, D. Delcourt, A. Doressoundiram, S. Erard, R. Gladstone, M. Grande, D. Hunten, W. Ip, V. Izmodenov, A. Jambon, R. Johnson, E. Kallio, R. Killen, R. Lallement, J. Luhmann, M. Mendillo, A. Milillo, H. Palme, A. Potter, S. Sasaki, D. Slater, A. Sprague, A. Stern, and N. Yan. PHEBUS: A double ultraviolet spectrometer to observe Mercury's exosphere. *Planetary and Space Science*, 58(1-2):201–223, 2010.

[15] Antoine Colin and Isabelle Puaut. A modular & retargetable framework for tree-based WCET analysis. In *ECRTS*, pages 37–44. IEEE Computer Society, 2001.

[16] CompCert development team. The CompCert formally verified compiler, version 1.13, 2008-2013. http://compcert.inria.fr.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[18] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[19] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - a software analysis perspective. In *SEFM*, pages 233–247, 2012.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[21] The Coq development team. *The Coq proof assistant reference manual*. INRIA, 2010. Version 8.3.

[22] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. Understanding integer overflow in C/C++. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 760–770. IEEE, 2012.

[23] Robert Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[24] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, pages 163–174. IEEE Computer Society, 2000.

[25] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Division of Computer Systems, 2003.

[26] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET bounds by abstract execution. In *Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011)*, 2011.

[27] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *WCET*, volume 6 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[28] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010.

[29] Christian Ferdinand and Reinhold Heckmann. aiT: worst case execution time prediction by static program analysis. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 377–384. Kluwer, 2004.

[30] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[31] Ricardo B. França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *PPES*, volume 18 of *OASICS*, pages 59–68. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011.

[32] Keith Gallagher and David Binkley. Program slicing. In *Frontiers of Software Maintenance, FoSM 2008*, pages 58–67. IEEE, 2008.

[33] Stéphane Glondu. *Vers une certification de l'extraction de Coq.* PhD thesis, Université Paris Diderot, 2012.

[34] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In Björn Lisper, editor, *WCET*, volume 15 of *OASICS*, pages 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[35] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, pages 57–66, 2006.

[36] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In Antoine Miné and David Schmidt, editors, *SAS*, volume 7460 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2012.

[37] Raimund Kirner. The WCET analysis tool CalcWcet167. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (2)*, volume 7610 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012.

[38] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Communications of ACM*, 53(6):107–115, 2010.

[39] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In Edmund M. Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2011.

[40] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing: on-demand feasibility refinement for proven precise WCET-bounds. In Michel Auguin, Robert de Simone, Robert Davis, and Emmanuel Grolleau, editors, *RTNS*, pages 161–170. ACM, 2013.

[41] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 389–392. ACM, 2007.

[42] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In Manuel V. Hermenegildo and Germán Puebla, editors, *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2002.

[43] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[44] Leonard Lensink, Sjaak Smetsers, and Marko C. J. D. van Eekelen. Generating verifiable Java code from verified PVS specifications. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 310–325. Springer, 2012.

[45] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

[46] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[47] Xavier Leroy. The CompCert C language. http://compcert.inria.fr/man/manual005.html, 2013.

[48] Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[49] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[50] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, pages 136–146. IEEE Computer Society, 2009.

[51] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 2011.

[52] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA*, pages 161–166. IEEE Computer Society, 2008.

[53] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[54] Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In Ranjit Jhala and Atsushi Igarashi, editors, *APLAS*, volume 7705 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2012.

[55] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[56] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[57] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[58] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[59] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound - a conceptually new tool for worst-case execution time analysis. In Raimund Kirner, editor, *WCET*, volume 8 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[60] Ganesan Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, 1999.

[61] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[62] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *WCET*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[63] Leanna Rierson. *Developing Safety-Critical Software*. CRC PressINC, 2013.

[64] Valentin Robert and Xavier Leroy. A formally-verified alias analysis. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2012.

[65] Xiaomu Shi, Jean-François Monin, Frédéric Tuong, and Frédéric Blanqui. First steps towards the certification of an ARM simulator using CompCert. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2011.

[66] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[67] Daniel Wasserrab and Andreas Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2008.

[68] Mark Weiser. Program slicing. In Seymour Jeffrey and Leon G. Stucki, editors, *ICSE*, pages 439–449. IEEE Computer Society, 1981.

[69] Freek Wiedijk. Comparing mathematical provers. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003.

[70] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[71] Reinhard Wilhelm and Daniel Grund. Computation takes time, but how much? *Communications of the ACM*, 57(2):94–103, 2014.